(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(54) Title: A DATA STRUCTURE, MEMORY ALLOCATOR AND MEMORY MANAGEMENT SYSTEM
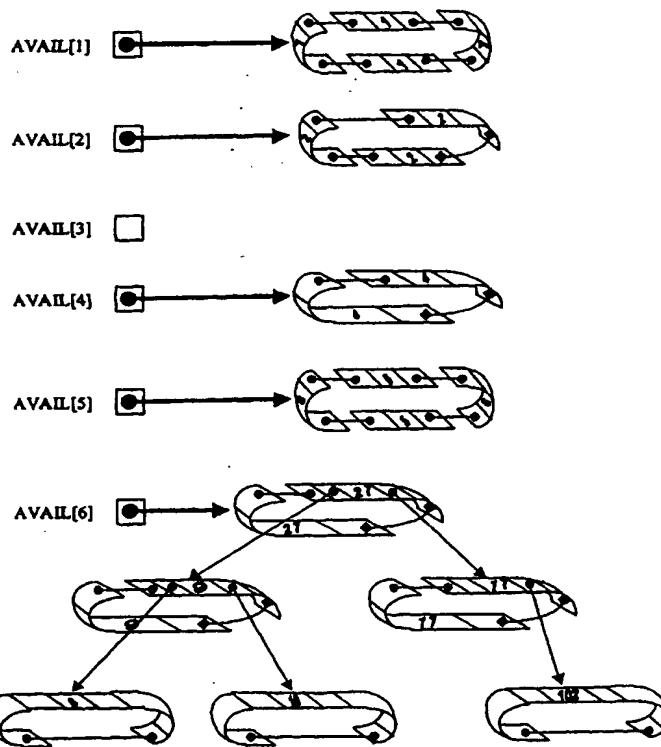
(57) Abstract: We present a Best Fit allocator for dynamic memory management. Portions of the memory that are presently unused are call free cells, and each free cell has a size. The allocator uses a bitmap which, for each number of predetermined sizes, indicates whether free memory cells of that size exist. It also employs a second data array with an entry for each of the predetermined cell sizes. When one or more free cells of a given size exist, the corresponding entry of the data array is a pointer to one of those free cells. The free cells themselves contain pointers to other free cells of the same size, or to free cells that are slightly smaller or larger. The allocator is scalable, in that the worst-case behaviour is independent of the size of the heap, and is independent of the number of free cells and of the number of cells already in use for memory storage. It is also incremental and non-disruptive, in that each memory operation (including splitting and coalescing of free cells) is guaranteed to complete within a small bounded time. We also present a novel collector and a priority queuing mechanism that operate on principles similar to those of the allocator.

WO 02/31660 A2

Published:

— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

A Data Structure, Memory Allocator and Memory Management System

Field of the invention

5

The present invention relates to storage of data in a
data storage space supported by one or more *data storage
devices,* and in particular to a data structure for
storing data within the *data storage space.* The invention

10   further relates to a data storage method using the data
structure, to a memory freeing method using the data
structure, to a data sorting method using the data
structure, and to apparatus arranged to perform any of
the above methods.

15

Background of the invention
Many of the tasks performed by modern computer systems
require the storage of large quantities of data using a
wide variety of *data storage devices.* In many of these

20   tasks the total quantity of the data to be stored is not
known in advance, and neither is the size of the
individual packets of data which are sent for storage.
Furthermore, the time for which data must be stored
varies widely according to the nature of the

25   *computational task* and the data itself.

For example, consider a computer system which is used for
image enhancement. The computer system may contain a bank
of original stored images of varying sizes which are held

30   in a relatively long term storage system until it is time
for one of those images to be extracted for enhancement.
At that time the extracted image may be copied (or
transferred) to a second memory device, and stored there
for a relatively short time scale while an enhanced

version of the image is prepared. During the image
enhancement procedure itself, calculation steps will be
performed in which relevant data (for example Fourier
data) is successively generated, stored in a third memory
5   device, extracted from that memory device for use, and
then deleted from that memory device. These operations
may occur on a yet shorter time scale: for example in the
time taken to process a small section of the image. The
amount of data stored at any moment will vary, and may
10   not be determined by (or even related to) the size of the
original image.

Furthermore, if the image enhancement algorithm is
written in a high level programming language, there will
15   inevitably be many steps of data storage and data
deletion performed by the operating system of the
computer of which the user is unaware. Many of these data
storage and deletion operations occur on a yet shorter
time scale, and involve varying quantities of data.
20
With this example in mind, let us at once clarify the
terminology used in this document.

The term "computational task" will be used here to
25   include any computing operation performed automatically
or semi-automatically by a processor (which may be a
single physical unit, or alternatively a distributed
processor made up of a plurality of physically separate,
or even spatially distant, units). For example,
30   "computational task" here includes image or sound
generation or processing, processing of business or other
financial data, text processing (for example, word
processing or translation), etc.

The term *"data storage device"* includes any device that
is capable of storing data, including devices which are
internal or external to the unit including the processor.
"Data storage device" thus includes a magnetic storage
5   device (e.g. a hard-disk drive or floppy disk), an
optical storage device (e.g. a CD storage device, such
as read/write CD disk system), an electronic memory
device (such as an internal or external RAM memory), and
even a *data storage device* which is a portion of an
10   integrated circuit which also carries some or all of the
processor itself.

One or more *data storage devices* may together define a
"data storage space". For example, the external RAM
15   memory of a conventional PC is a *data storage space,*
normally supported by one or more integrated memory
chips. The hard drive supports a second *data storage
space.* And both of these *data storage spaces* are parts of
a single *data storage space* which is the total *data
20   storage space* of the PC.

The term *"data item"* is used herein to include any data
which is intended to be stored in the *data storage space.*
A *data item* may have any length (for example use of the
25   plural word "data" does not imply that the *data item*
consists of more than one bit) and any significance. For
example, it may be a program (in any language), or any
data to be subjected to computer processing, or which
arises during computer processing.
30

The term "*data array*" is used to mean a section of a *data
storage space* that is composed of one or more "*fields*"
having a logical relationship to each other. A "*field*"
may be of any *size,* corresponding to one or more *data*

items. No *size* limitation on a *field* is implied: for
example, the *size* of a particular *field* may be one bit,
one byte, one word, or longer. No limitation on the
location of a *field* is implied: for example, a *data array*
5   may be composed of *fields* that are contiguously located
in the *data storage space,* or may be composed of *fields*
that are located at discontiguous locations on different
data storage devices in the *data storage space.*
Similarly, no limitation on the location of the contents
10   of a *field* is implied: for example, a *field* may be
composed of *data items* that are contiguously located in
the *data storage space,* or may be composed of *data items*
that are located at discontiguous locations on different
data storage devices in the *data storage space.* The
15   logical relationship between the *fields* may be a linear
one (for example each *field* may correspond to a
respective section of a *data storage space,* and the order
of the *fields* in the *data array* may correspond to the
order of those sections within the *data storage space* (
20   e.g. the order of the addresses», but any other logical
relationship is also possible. For example, as discussed
below in detail, the logical relationship between the
fields may be hierarchical (e.g. the *data array* may be
logically equivalent to a tree-like structure of *fields).*
25

Data items are either pre-existent within a *data storage*
space or are stored by being written to locations within
the *data storage space.* In a conventional computer
system, selecting the locations for storage within the
30   *data storage space* is one of the tasks of a unit called
an "*allocator*" (which may be one or more dedicated
components, or implemented in software, e.g. as one of
several functions performed by a CPU). The allocator
selects locations in logical units of a "*cell*": a "*cell*"

is a data array .The *computational task* is performed by a
unit called a *"mutator"* (which may be one or more
dedicated components, or implemented in software, e.g. as
one of several functions performed by a CPU). As the
5 *mutator* performs its computational task, *data items* are
continually being written into the *data storage space*,
and deleted. Thus at any moment the *data storage space*
contains *cells* which are being used by the *mutator* for
storing data (such an area is known as a *"live cell")*,
10 and *cells-which* are not being used by the *mutator* for
storing data (such an area is known as a *"free cell")*. A
live cell may also become *a free cell* without any
deletion of the *data items* within that *cell:* this is
either achieved through express communication between the
15 *mutator* and the *allocator* or through a process of usage
analysis by a unit called the "garbage collector", or
simply the *"collector"* (which may be one or more
dedicated components, or implemented in software, e.g. as
one of several functions performed by a CPU). A *cell*
20 which is not live but which contains *data items* that are
still useful to the *allocator* or to the *collector* is
called a *"zombie cell",* or simply a *"zombie"*. In general
terms, data storage using *cells* is referred to as II *heap
storage"*. The total *size* of a *data storage space* which is
25 available for *heap storage* will here be referred to as
the quantity HEAPSIZE of that *data storage space*. The
HEAPSIZE for a *data storage space* may vary during the
course of the *computational-task* through the addition or
deletion of sections of memory: such sections of memory
30 are called *"chunks"* and the heap storage so implemented
is called a *"chunked heap"*.

For most *allocators,* a *cell* includes zero or more *fields*
for the data to be stored (i.e. a

data item) and zero or more *fields* for additional data
which is used for reference purposes
(e.g. for the purpose of memory management itself). The
latter form of data is here referred to as *"reference*
5   *data"*. For example, for a one-dimensional *data storage*
space, each live *cell* may store *reference data* that
includes the address in the *data storage space* of one of
the *fields* of the *cell.* Further *reference data* may
include a *field* indicating that the cell is in fact live,
10   a *field* indicating whether neighbouring *cells* in the *data*
storage space are free or live, and *afield* indicating the
locations of other *live cells* within the *data storage*
space. For the avoidance of doubt, a *free cell* or a
zombie cell may contain *reference data* even though it may
15   not contain a *data item.*

We define the" *size"* of a *cell* to mean the space within
it which is useable for storing
a *data item.* Thus, the" *size"* of a *cell* roughly
20   corresponds to the amount of the *data storage space*
occupied by the *cell,* but the correspondence need not be
exact due to the possibility of *reference data. Size* is
an integer multiple of a fundamental data unit.
Preferably, each *cell* will store *reference data* including
25   the *size* of said *cell.*

When a *data item* is to be stored, how should an *allocator*
search for a suitable *free cell* to store it in? One
simple strategy is for the *allocator* to search through
30   the *data storage space* (e.g. in address order) for the
first *free cell* large enough to store the *data item,* and
store it there. This is known as a" *First Fit"* method.

An alternative would be to search all available free
cells and see if there is one *cell* of exactly the right
size to store the *data item*. If there is not, the
allocator can then search for the smallest *cell* that is
5   large enough to store the *data item*. This is called a
"*Best Fit*" criterion. In the case that the *free cell* is
larger than necessary to store the *data item,* the part of
the *free cell* which is not used for storing the *data item*
(if it is above a predetermined *size*) may become a new
10   *free cell.*

A further alternative is to search all available *free
cells* to find the largest one, and store the *data item*
there. Again, in the case that this *free cell* is larger
15   than necessary to store the *data item,* the part of the
free cell which is not used for storing the *data item*
becomes a new *free cell.* This is called a" *Worst Fit"*
selection criterion. The rationale of the " *Worst Fit"*
selection criterion is that the part of the *free cell*
20   which is not used for storing the *data item* will tend to
be large, and thus" *Worst Fit"* will produce relatively
large new *free cells.*

Which of the above criteria is optimal generally depends
25   upon the *computational task* in question (e.g. the
distribution of the *data items,* both in time and size,
and the order of allocation and freeing of *cells).*
Preferably, a selection criterion should not lead to
"*fragmentation*" (that is, the creation of a very large
30   number of *small free cells).* Also, preferably the time
taken by the *allocator* should be "*non-disruptive"* (that
is, it should never, even in a worst case, be
unacceptably large) and should be "*scalable*" (that is, it
should be independent of the value of HEAPSIZE).

A survey of the field of *allocators* is given in the paper
"Dynamic storage allocation: a survey and critical
review" by Paul R. Wilson, Mark, S. Johnstone, Michael
5   Neely, and David Boles (in the Proceedings of the
International Workshop on Memory Management, vol 986 of
Lecture Notes in Computer Science, Kinross, Scotland,
September 1995, Springer- Verlag). As pointed out there,
the efficiency of an *allocator* in storing memory in a
10 *data storage space* has major cost implications. Assuming,
for example, that there are 100 million PCs in the world,
each having 10 MBytes of memory at $30 per Byte. If just
one fifth is used for heap-allocated data, and one fifth
of that is unnecessarily wasted, the cost is over a
15 billion dollars.

A further survey of the field was given by Thomas A.
Standish in "Data Structure Techniques", Addison-Wesley,
1980.It is now generally accepted that *Best Fit*
20 allocation is one of the best strategies in practice
(i.e. in most cases, on real loads, compared with other
general approaches such as *First Fit)* for reducing the
fragmentation that occurs in dynamic memory management of
variable-sized *cells*. For example, Shore ("On the
25 external storage fragmentation produced by first-fit and
best-fit allocation strategies", Communications of the
ACM, 18(8): 433-440, August 1975) found that *Best Fit* and
address-ordered *First Fit* allocation work equally well,
and Wilson and Johnstone ("Real-time non-copying *garbage*
30 *collection",* in Eliot Moss, Paul R. Wilson and B. Zorn,
editors, OOPSLNECOOP;'93 Workshop on *Garbage collection*
in Object-Oriented Systems, October 1993) present
evidence that *Best Fit* is "one of the best policies" for
reducing fragmentation.

However, it is widely believed that *Best Fit* allocation
is not *scalable* because the time taken to search for the
best-fitting free cell must always be related to

5   HEAPSIZE. In the best case, it is assumed that the use of
sorted trees can reduce the searc4ing so that it scales
with

the logarithm of HEAPSIZE.

10 A known *allocator* that uses a *Best Fit* criterion, called
"Fast-fit" (described by Standish), will now be described
in detail with reference to Fig. 1. This diagram shows a
data structure for a memory with *free cells* of *sizes*
1,1,1,1,2,2,2,4,4,5,5,5,5,8,9,9,10,27 ,27, 77, 77 and

15 102. Each *free cell* contains at each end a pointer (shown
in grey on the figure), pointing to another *free cell* of,
the same *size* (where one exists) or to the other end of
itself (if no other *free cell* of the same *size* exists).
Thus, the *free cells* contain enough information to define

20 "rings" of *free cells* of the same *size*. The *allocator*
also uses a *data array* AVAIL having six *fields*. AVAIL(1)
is used to store the location of one of the four *free
cells* of *size* 1, AVAIL(2) stores the location of a *free
cell* of *size* 2. There are no *free cells* of *size* 3, and

25 this fact is indicated in AVAIL(3). AVAIL(4) and AVAIL(5)
store the location of a *free cell* of *size* 4 and 5
respectively. *Cells* above *size* 5 are treated differently.
Specifically, for each ring, a representative *cell* is
selected, and the representative *cells* are sorted into an

30 AVL tree.
To store a *data item* of *size* n, the following procedure
is used.

• If n less than or equal to 5, and AVAIL(n) is not
empty, the *data item* is stored in *the free cell*

indicated by AVAIL(n). If AVAIL(n) is empty, we
search for the smallest j (larger than n) for which
AVAIL(j) is not empty. If j is less than or equal to
5, the *free cell* indicated by AVAIL(j) is split into
free cells of *size* n and (j-n), and the *data item* is
stored in the *cell* of *size* n.

- If n is greater than or equal to 6 (or in the case
  in the last paragraph, if no j less than or equal to
  5 can be found) we use *n* as a search key to search
  the AVL tree pointed to by AVAIL(6), and locate a
  ring of *cells* whose *size* s is the smallest that is
  also greater than or equal to n, and if s does not
  equal n then split it to form two *free cells* of *size*
  n and (s-n). The *data item* is stored in the *free*
  cell of *size* n thus found or created.

- In all cases where an appropriate *free cell* is
  found, that *cell* is detached from its ring, with the
  pointers in the remaining *cells* being adjusted
  accordingly.

- Whenever a *cell* is split, leaving a remaining *free*
  cell of *size* (j-n) or (s-n), we check
  to see whether the neighbouring *cell* to the
  remaining *free cell* is free or not. If it is, we
  detach the neighbour from its ring and coalesce the
  new *free cell* with that neighbour to form a
  largerfree cell. The new *free cell* is inserted into
  the structure shown in Fig. 1.

The present inventor has investigated this scheme and has
noticed a number of features of it. In general the time
taken to identify a *free cell* for storing a *data item*
rises (in the worst case) as the number of layers of the
hierarchical tree. *Since free cells* may exist up

to any arbitrary *size* less than HEAPSIZE, and each ring
in the tree may contain a single *cell*, and the tree has
binary division, the number of layers in the worst case
is of the order of log2(HEAPSIZE'/'). Thus in the worst
5   case, the time taken for a memory storage task increases
with increasing HEAPSIZE of the *data storage space*.

The present inventor has devised a variation of the Fast-
fit algorithm in which it is ensured *all free cells* have
10   a *size* less than or equal to a maximum MAXFIELDS (the
data
storage space may, for example, be generated with all
cells less than or equal to this *size,* and coalescing of
two *cells* may only be carried out in the case than a *cell*
15   larger than
MAX FIELDS is not created). In this case, the time taken,
for a memory storage operation is in the worst case log2
(MAXFIELDS), which is more acceptable since MAXFIELDS is
not related to HEAPSIZE and may be very much smaller than
20   the square root of HEAPSIZE.

However, this estimate assumes that the AVL tree is
approximately perfectly balanced. This would not be the
case, for example, in Fig. 1 if the two *cells* of *size* 77
25   and one *cell* of *size* 102 were used up. As the tree
becomes less balanced finding *cells* of a given *size* takes
longer compared to the number of rings, and in the worst
case, the time taken to find a *free cell* scales with the
number of rings. To ensure that this does not occur, a
30   complex and time consuming occasional "re-balancing"
algorithm would be required (re-balancing of an AVL tree
is called "rotation"). While this operation is carried
out, data storage is not possible, and thus the re-

balancing may cause unacceptable disruption of the
computational task.

Any *allocator* gradually uses up the *free cells* in the
5 *data storage space,* so, for memory storage to be
sustainable, there must be a mechanism for generating new
free cells. That is, when it is no longer required to
store the data in a *live cell,* there should be a
mechanism for converting the *live cell* into a *free cell*
10 (i.e. for "freeing" *cells).*

In some systems, the freeing of memory is decided by the
programmer. However, it
is increasingly common for programming systems to offer
15 automatic dynamic memory management including automatic
freeing of *cells* that are no longer being used by the
mutator; such cells are called " *garbage cells"* .This is
referred to here as " *garbage collection",* or simply
"collection", and is performed by the *collector* (as
20 previously defined).

In the implementation of functional programming languages
in particular, automatic detection and freeing of *garbage
cells* is required. Preferably, *collection* should be
25 *scalable,* and *non-disruptive.* Such *collection* overheads
(which lead to short memory-management pauses due to
allocation or *collection)* are vital for interactive and
real-time applications.

30 Baker's incremental copying technique (Henry G. Baker,
"List processing in real-time on a serial computer".
Communications of the ACM, 21(4); 280-94, 1978) might be
expected to form the natural basis for a *collector* with
the required behaviour. However, the worst-case cost of

Baker's read barrier is potentially very high and very unpredictable.

Another sort of *collection* system (see G. E. Collins, "A method for overlapping and erasure of lists". Communications of the ACM, 3(12):655-657, December 1960) is known as *"reference counting"* (i.e. counting the number of *live cells* which point to a particular *cell* - for each *cell*, this is called the *"reference count"* for that *cell*). The *reference count* for each *cell* is normally held as part of the *reference data* for that *cell*. The advantages of *reference counting* are:

- it is naturally incremental (it is performed in small steps interleaved with computation);
- it is naturally distributed and can improve locality of reference.
- it is easy to implement;
- there is no root-set to be scanned (an important determinant of real-time performance);
- the overheads are predictably related to actual *mutator* activity rather than to the size of the heap or the numbers of *live cells* (many of which may be inactive for long periods);
- its behaviour does not degrade with increasing heap occupancy;
- it is decoupled to a large degree from the *mutator;* synchronisation with the *mutator* is not required, and (unlike incremental tracing *collectors)* the *collector* is not fooled by *mutator* activity -the algorithm is therefore simpler.

Consider the following three memory operations:

    1. *cell* allocation (and issuing of a pointer to the
mutator)

    2. pointer copy

    3. pointer deletion

Using *reference counting,* the first two of these
operations may be achieved with constant-time *garbage
collection* overhead. However, a complication arises after
a *garbage cell* has been identified (always as the direct
result of pointer deletion). If a *cell's reference count*
drops to zero, it is garbage; before returning the
garbage cell to the *allocator's* data structure for
managing *free cells,* it is necessary to inspect each of
this *cell's fields* and delete any pointers held therein.
We call this action 'freeing" *a field.* The deletion of a
pointer in this way might lead to another *cell* being
identified as garbage, so potentially leading to a
cascade of deletions. The time taken to service a
deletion cascade is unpredictable and may be large. It is
therefore important that such cascades are avoided.

This problem is well-known and solutions have previously
been proposed, for example by *I.* Wiezenbaum ("Symmetric
list processor", Communications of the ACM, 6(9): 524-
544, September 1963).

- When a *cell* is identified as garbage it is placed on
  a "to be collected" data structure - these are
  zombie cells, as previously defined;

- The *fields* of *zombie cells* are freed incrementally,
  interlaced with other *mutator* activity;

- When all *fields* in a *zombie* have been freed, the
  cell becomes free (it is passed to the *allocator).*

One option is to inspect *zombie fields* just prior to each
cell allocation (since at this point we know that the

mutator requires more memory). If the freeing of a *zombie field* causes a pointer deletion, thereby causing another *cell's reference count* to become zero, then that *cell* simply becomes a *zombie* -the *fields* in the subsequent

5   *cell* are not inspected and no cascade occurs. This simple procedure eliminates cascades but introduces a further tension:

- If we guarantee always to free more *zombiefields* (if they exist) than the number of *fields* required for

10   the *cell* allocation, the *collector* always makes progress *(collection* exceeds allocation whilst there is garbage to be collected);

- If we guarantee always to free an exact number of *zombie fields* before each allocation, the *garbage*

15   *collection* overhead for *cell* allocation is constant (given a fixed MAXFIELDS limit).

In a system with variable-sized *cells,* it is not possible to guarantee both that the *collector* makes progress and that the *collection* overhead per allocation is a small

20   constant.

At the point of *cell* allocation, a *zombie* cell-should be freed. But which *zombie* should we choose?

- If we always free the largest *zombie* first, the resultant *free cell* can be used to satisfy many more

25   allocation requests than a small *cell;*

- By contrast, Last-In-First-Out (LIFO) freeing (i.e. the most recent) encourages depth- first freeing of cascades and might lead to better cache performance;

- However, it may be preferable to coalesce older

30   *cells* before younger *cells* using First- In-First-Out *(FIFO) freeing,* in order to improve fragmentation behaviour.

The choice of strategy affects the overheads for *cell* allocation. Both LIFO and FIFO *zombie* freeing can be achieved with constant time overhead per operation using a stack or queue, whereas largest-freeing requires that

5   the largest *zombie* be found. This is a search comparable in nature to the search described above in relation to *free cells,* and thus exhibits the same problems of scaling and possible disruption.

10   Summary of the invention

The present invention seeks to address one or more of the above problems, and to
provide a novel and useful data structure.

15   Preferably, the present invention seeks to provide a data structure that-makes possible a memory management system which is *scalable, non-disruptive,* and employs *Best Fit* allocation together with *non-disruptive* coalescing.

20   Preferably, the present invention makes it possible to achieve a *(chunked)* heap of variable-sized *cells* to be managed such that:

- *Fragmentation* is minimised, using *Best Fit* allocation (including cell-splitting and coalescing *offree cells).*

25 - The system is *scalable,* i.e. has memory-management overheads (in terms of memory accesses) being independent of the numbers *offree cells* or *live cells,* and independent of the *size* of the heap (which, with a *chunked* heap, may grow at runtime).

30 - Memory management is *non-disruptive* (for example, there are no "embarrassing pauses" whilst *free cells* are coalesced), and the three operations of *cell* freeing, *cell* allocation and *cell* coalescing each have a small bounded overhead in terms of memory accesses.

In a first aspect, the invention proposes in general
terms that a data structure includes a first *data array*
which indicates the existence of *cells* of a predetermined
type (e.g.. *free cells)* and of a predetermined *size,* and
a second *data array* which indicates the location of such
cells within the *data storage space.*
Specifically, in this first aspect the invention provides
a data structure comprising:

10 • a *data storage space;*

• a first *data array* which, for each of one or more *cell
sizes,* stores data indicating whether or not the *data
storage space* includes at least one *cell* of a certain
type and of the respective *size;* and

15 • a second *data array* which, for any said *size* for which
at least one *cell* of the determined type exists,
indicates a location of at least one *cell* of that *size*
and type within the *data storage space.*

20 The first and second *data arrays* may each be sections
(which may be allocated statically before the start of
the computational task, or dynamically during the
computational task, or comprise both statically-allocated
and dynamically-allocated parts) of said *data storage*
25 *space,* or sections of one or more additional *data storage
spaces.* Though logically separate, the first and second
data arrays may be merged to facilitate implementation.
The "type" of *cells* may be *free cells,* or alternatively
live cells. However, the term "type" is not limited in
30 this respect. For example, a "type" of *cells* may be *cells*
which, irrespective of whether they are live or free,
possess some additional characteristic, such as including
certain data. Preferably, the *cell* types are
predetermined, but alternatively it would be possible for

them to vary (e.g. be regularly reset) as part of an optimisation of the data structure in relation to the *computational task* in which it is employed.

One type of *cell* to which the invention is
5 applicable, for example, is *cells* which do
not contain *data item(s)* that are useful to the mutator
but do contain pointers (to other *cells)* which may be
useful to either the collector or the allocator. For
example, *zombie cells* discussed above.

10   In the case that the type of *cells* is *free cells*
(i.e. portions of the *data storage space* which are not
presently being used by the *mutator* -for storing data),
when it is desired to store a data unit of a certain
size, the first *data array* can be used to identify a
15 suitable *cell size* for storing it (e.g. the smallest
cell size which is at least as large as the data unit and
for which at least *onefree cell* exists), and the second
data array can be used to locate a *free cell* of that
size, so that the data unit can be inserted into it.

20   Preferably, said predetermined *cell sizes* are all no
more than a predetermined maximum *cell size* (MAXFIELDS),
which is independent of the total *size* of the *data
storage space* (HEAPSIZE). For example, said predetermined
cell sizes may be all *cell sizes* up to the predetermined
25 maximum *cell size.* This means that the *size* of both the
first and second *data array* may depend upon MAXFIELDS but
be unrelated to HEAPSIZE.

Preferably, the first *data array* may be selected so
that for a *cell size* n it is possible to determine the
30 smallest (integer) value of p such that p>n and there
exists at least one *cell* of the predetermined type of
size p, within a time which scales with log (MAXFIELDS).

Alternatively, or additionally, the first *data array*
may be such that for a *cell size* n it is possible to

determine the largest (integer) value of p such that p<n
and there exists at least one *cell* of the predetermined
type of *size* p, within a time which scales with log
(MAXFIELDS).

5   For example, the first *data array* may be a
hierarchical *data array,* having:

● a first level which includes one or more first
        fields, each first *field* indicating for a respective
        size of *cell* the existence or otherwise of at least
10   one *cell* of that *size,* said first *fields* being
        divided into one or more groups, each group
        representing a respective range of *cell sizes;* and

● a second level which includes one or more second
        fields, each second *field* indicating for a
15   respective said group the existence or otherwise of
        at least one *cell* in the respective range of *cell*
        sizes.

We will refer here to the first layer as the "lowermost"
layer, and to the second layer as a "higher" layer. We
20 will use the names "layer" and "level" synonymously. We
will say that a particular *cell size* is *"related"* to any
field of the first *data array* when that *field* either (i)
contains information about the existence of *cells* of that
size, or (ii) contains information about another *field*
25 which is itself *related* to the particular *cell size.* When
for a given *cell size* there is at least one *cell* of the
predetermined type having that *cell size,* we say that
that *cell size* is "occupied". Similarly, when a *field* in
the first *data array* indicates that one of the *cell sizes*
30 *related* to it is occupied, we will say that that *field* is
"occupied".

        Preferably, the first *data array* contains further
layers "above" the second layer. In each i-th layer there
are one or more *fields.* Unless the i-th layer is the top

layer, the one or more *fields* are divided into one or
more groups, each group representing a range of *cell
sizes,* and in the (i+1)-th layer there is *a field* for
each respective group of the i-th layer. Thus, any
5   particular *cell size* is, in each layer, *related* to a
single respective *field.* The existence of at least one
cell of that particular *cell size* determines information
in all *fields related* to it, in all layers of the *data
array.*

10   If all groups contain two or more fields, at each
level the number *of fields* is at most half the number of
fields in the layer below. To be precise, if each *field*
in an (i+1)-th layer represents a group *of p fields* in
the i-th layer, the (i+1)-th layer contains p times as
15   many *fields* as the i-th layer. The largest *cell size*
represented by the first *data array* is MAXFIELDS, so the
total number of layers scales as log p (MAXFIELDS). This
has the effect that a search within it for a *cell size*
which is occupied, takes a time which scales as log p
20   (MAXFIELDS).

        We will now give a single example of how this search
may be carried out:

25   Suppose that for a given (unoccupied) *cell size* n we wish
to find the lowest occupied *cell size* higher than n. To
do this we begin at the lowest level of the table with
the *field* related to n. We then ascend the table layer-
by-layer.

30   • In each layer including the lowest we determine
        whether, in the group including the *field related* to
        cell size n, there is at least one *occupiedfield*
        representing *cells* larger than n.

- If so, then we select the smallest such *occupiedfield,* and move back down the layers, at each layer selecting the occupied *field* which is *related* to the selected *field* in the layer above and which represents the smallest *cells. The cell size* we arrive at in the lowest layer-is the one sought.

- Otherwise, we go up one layer higher.

- If we reach the top layer, we check to see whether in that top layer there is at least one occupied *field* representing *cells* larger than n.

- If so, then we select the smallest such occupied *field,* and move back down the layers, at each layer selecting the occupied *field* which is *related* to the selected *field* in the layer above and which represents the smallest *cells. The cell size* we reach in the lowest layer is the one sought.

- Otherwise, no such *cell* exists.

Preferably, in each layer, each group represents a number *of fields* in the layer below which is a multiple of eight (e.g. eight, sixteen or thirty-two). Occupancy or otherwise of a *field* may be represented by a binary digit. In this case, the group of *fields* may conveniently be represented as an integer number of bytes. In this case, for many computer systems, the operations required for searching the tree can be carried out significantly more quickly (in real time), because the *data array* can be simply a bitmap.

If there are *32fields* per group, then it is possible to achieve the search in a time of O(log32 (MAXFIELDS)). IfMAXFIELDS=32k, log32 (MAXFIELDS) is just 3, so the first *data array* has only three levels.

Although, as mentioned above, the predetermined *cell sizes* may be all *cell sizes* up to the value of MAXFIELDS, this is not in fact a necessary feature of the invention. A further option within the scope of the invention is to
5 treat *cells* in different ways according to their *size*. For example, it may be the case that due to *fragmentation* there will always in practice be a large number of very small *cells*, so that the first *data array* is not needed for such *cells*. Thus, it may prove more efficient for the
10 first *data array* only to be used for *cells* having a *size* which is at least a predetermined minimum *size* (here called "MINFIELDS"), and for *cells* smaller than MINFIELDS to be indexed differently. For example, the very small *cells* might be treated in a manner similar to the Fastfit
15 algorithm described above. That is, for *cells* smaller than MINFIELDS, the second *data array* could contain: (i), in the case of a *cell size* for which *cells* do exist, a pointer to such a *cell*, or (ii) in the case of a *cell size* for which *cells* do not exist, data indicating that
20 fact.

Thus, for example, in the case that the *cells* are *free cells*, and it is desired to locate a very small *free cell* (i.e. below the predetermined *size*) to store a small data unit, this could be done using the second *data array*
25 only. Provided that MINFIELDS is a value that is unrelated to either HEAPSIZE or MAXFIELDS (that is, it does not scale with HEAPSIZE or MAXFIELDS), the time scaling of the overall algorithm may be unaffected.

The existence of MAXFIELDS is not in fact a
30 necessary feature of the invention. A further option within the scope of the invention is to allow the first and second *data arrays* to contain *fields* for all *cell sizes* up to HEAPSIZE (which may vary). This option may be further optimised within the scope of the invention to

allow the first and second *data arrays* to contain fields
for all *cell sizes* up to the largest *cell size* yet
allocated at any given time during the performance of the
computational task and for the size of the first and

5   second *data arrays* to be modified dynamically (including
extension, contraction and re-balancing as necessary)
during the *computational task*.

The invention has been explained above using a first
data array which indicates the existence or otherwise of

10   a *cell* of an exactly predefined *size*. However, all of the
above concepts are straightforwardly applicable to a case
in which the *cell sizes* are in fact binned (i.e.
conceptually divided into ranges), and the first *data
array* is in fact used to indicate the existence of a *cell*

15   within a certain range.

Thus, the first aspect of the invention may
alternatively be expressed as providing a data structure
comprising:

20   • a *data storage space;*

• a first *data array* which, for each of one or more *cell
size* ranges, stores data indicating whether or not the
data storage space includes at least one *cell* of a
certain type within the respective *size* range; and

25   • .a second *data array* which, for any of said *size* ranges
for which at least one *cell* of the determined type
exists, indicates a location of at least one of those
cells within the *data storage space*.

In the case that the *cell size* ranges are each only

30   one unit wide, this reduces to the expression of the
first aspect of the invention given earlier. However,
alternatively, any *cell size* range may be wider than one
data unit (so that it includes two or more possible *cell*

sizes). The largest *cell size* range may be defined as all *cells* up to a predefined *size* (which corresponds to MAXFIELDS, or to HEAPSIZE in the option that MAXFIELDS is not used).

5   Preferably, the *cell size* ranges are predetermined, but alternatively it would be possible for them to vary (e.g. be regularly reset) as part of an optimisation of the data structure in relation to the *computational task* in which it is employed.

10   In either case, the width of a given *cell size* range may be a function of the lower limit of the *cell size* range. For example, the *cell size* range may for example be selected to be wider for higher *cell sizes,* since in that case the

15   inefficiency caused by the coarse graining of the *cell sizes* (i.e. the waste caused by replacing an "exact-fit", by an approximate fit) is only a small fraction of the *size* of the *cell.* This is in fact a further example of the option mentioned above, of treating *cells* of

20   different *sizes* differently.

      Preferably, the *cell* types are predetermined, but alternatively it would be possible for them to vary (e.g. be regularly reset) as part of an optimisation of the data structure in relation to the *computational task* in

25   which it is employed.

      As mentioned above, all preferable features of the data structure according to the first aspect of the invention are applicable within the scope of the invention to the case in which the *cell sizes* (whether

30   predetermined or varying)-are generalised to *cell size* ranges. However, for simplicity of terminology, the following explanation is given on the basis that the *cell sizes* are predetermined as exact values (i.e. not ranges). Similarly, for simplicity of terminology, the

following explanation is given on the basis that the *cell* types are predetermined.

The data structure preferably includes *a field* which stores the location of at least one of: (i) a *cell* of the predetermined type having the greatest *size* (among the predetermined *sizes)* for which *cells* of the predetermined type exist *(this field* is herein called "MAXP"), or (ii) a *cell* of the predetermined type having the smallest *size* (among the predetermined *sizes)* for which *cells* of the predetermined type exist *(this field* is herein called "MINP").

MAXP is useful, for example, in the case that the predetermined *cell* type is *a free cell,* because it means that the *Worst Fit* method can be used for selection of a *free cell* to hold a *data item.* MAXP is also useful in the case that , the predetermined *cell* type is a *zombie cell,* because in that case it is easy to find the largest *zombie cell* to free.

Furthermore, in the case that the predetermined type of *cell* is *a free cell,* if it is determined (e.g. using the first *data array)* that no *cell sizes* equal to or less than n are occupied, then MINP points to *a free cell* which (among the extant *free cells)* is of the smallest *size* capable of storing a *data item* of *size* n.

Preferably, the *cells* of the predetermined type contain pointers to (i.e. data indicating the location of) each other. A pointer is a further example *of reference data.* For example, each *cell* may contain at least two such pointers.

Preferably, pointers of the *cells* of each predetermined *size* are such that, starting from any *cell* of any predetermined *cell size,* any other *cell* of the predetermined type and the same *cell size* can be

reached by moving successively from one *cell* to another according to the pointers.

Preferably, at least one *cell* of each predetermined *size* contains a pointer to one of the *cells* of the next higher occupied predetermined *cell size.* Preferably, at least one *cell* of each predetermined *size* contains a pointer to one of the *cells* of the next lower occupied predetermined *size.* If both of these conditions are met, then it is possible to move between all *cells* of the predetermined type having any of the predetermined *sizes* using pointer information alone.

For example, in the case that the predetermined type of *cell* is a *zombie cell,* it is possible to start at the *cell* indicated by MAXP, and move through all the *zombie cells* in an order in which the *size* of the *zombie cells* never increases (i.e. only decreases or remains the same), using the pointers. This possibility may be particularly useful for *garbage collection.*

The order of the path defined by the pointers through *cells* of the same *size* may encode information about the *cells* of that *size,* for example the age of the *cells* (i.e. the time since they were created).

For example, in the case of a path towards lower *cell sizes,* the *cell* of any given *size* which is indicated (pointed at) by a *cell* of the next higher *size,* may be the oldest *cell* of the given *size,* and the *cells* along the path towards the *cell* which points to the next lower *size* may be progressively younger. This age order makes it easy to locate old (or young) *cells* of a given type and *size* (for example, for conversion to another type of *cell).* The age order may be preserved by, whenever a new *cell* of the given *size* is created, changing the pointer in the youngest *cell* of the given *size* to point at the

new *cell,* and arranging for the new *cell* to contain a pointer to a *cell* of the next lower *size.*

Clearly, the scheme of the preceding paragraph may be varied within the scope of the invention, to provide a data structure in which the path leads from young *cells* to old *cells,* and/or to the case that the path leads from small to large *cells.*

Preferably, whenever a first *cell* of the predetermined type contains a pointer to a second *cell* of the predetermined type (of the same or different *size),* that second *cell* contains a pointer to the first *cell.* This has the effect that the path may be followed in either direction (i.e. towards either larger or smaller *cells).*

Although in principle the second data array may store the location of any .(or all) of the cells of the predetermined size (or size range), it is preferable that the (at least one) cell of each size (or size range) is selected according to a characteristic of the cell. For example, a further option within the scope of the invention is for the second *data array* to contain, for each *cell size* or *cell size* range, the locations of both the youngest and the oldest *cells* of the predetermined type and of the predetermined *size.* Alternatively, or additionally, the second *data array* may contain, for each *cell size* range, the locations of both the biggest and the smallest *cells* of the predetermined type and of the predetermined *size.*

A straightforward way of realising MINP is for the data structure to include at least one *cell* of the predetermined type (herein referred to as a" *sentinel cell")* which is of *size* MINSIZE or less (e.g. it might be of *size* zero) having a predetermined location in the *data storage space* and containing a pointer to a *cell* of

the predetermined type of the next higher *size*. In other
words, a pointer to the *sentinel cell* thus constitutes
the MINP *field.* Since the *sentinel cell* exists, a search
using the first *data array* for a *cell* of the

5 predetermined type of highest *size* lower than *size* n will
find the *sentinel cell* if no *cell* of intermediate *size*
exists, and the pointer in the *sentinel cell* leads to a
cell of the minimum *size* greater than n.

In principle, a *sentinel cell* with *size* MAXFIELDS

10 and a predetermined location
could additionally, or alternatively, be provided in the
data storage structure.

A further option within the scope of the invention
is for either the MAXP field or the

15 MINP field or both fields to be contained in the second
data array.

In fact, the concept of using pointers as described
above constitutes an independent second aspect of the
invention.

20 Specifically, in the second aspect the invention
proposes a data structure including:

• a *data storage space,* the *data storage space* comprising
one or more *cells* of a certain type, each said *cell* of
said type containing at least one pointer to the

25 location in the *data storage space* of another said *cell*
of said type; and

• *a field* which stores as one or more pointers the
location within the *data storage space* of at least one
of: (i) one of said *cells* of said type which is at

30 least as large as any other of said *cells* and of said
type, or (ii) one of said *cells* of said type which is
at least as small as any other of said *cells* and of
said type;

- said pointers being such that, starting from a *cell* of which the location is indicated in said *field,* any other of said *cells* can be reached by moving successively from one *cell* to another according to the

5   pointers in a path in which the *size* of the *cells* varies monotonically.

By "monotonically" is meant that the *size* of the *cells* in the path either (i) never falls

10   (e.g. in the case of starting with MINP), or (ii) never rises (e.g. in the case of starting with MAXP).

For example, at least one *cell* of each *size* may contain a pointer to a first one of the *cells* of the next higher (or lower) second *cell size,* and the pointers

15   within the *cells* of the second *size* may define a path from the first *cell* through all the other *cells* of the second *size* to the *cell* of the second *size* which contains a pointer to a *cell* of the next higher (or lower) *cell size.*

20   Preferably, as in the first aspect of the current invention, the *cell* types are predetermined, but alternatively it would be possible for them to vary (e.g. be regularly reset) as part of an optimisation of the data structure in relation to the *computational task* in

25   which it is employed.

The invention is defined above in terms of data structures. However, in third and fourth aspects, the invention in general terms respectively proposes using a

30   data structure according to the first aspect or second aspect of the invention in a memory management method. The data structure may have any of the preferable features described in detail above in relation to the first aspect or second aspect of the invention.

Specifically, in the third aspect the invention
provides a memory management method for controlling one
or more *data storage devices* which support a *data storage
space,* the memory management method including:

5   • deriving a first *data array* which, for each of one
          or more *cell sizes,* stores data indicating whether
          or not the *data storage space* includes at least one
          free cell of the respective *size;*

        • deriving a second *data array* which, for any said

10   *size* for which at least one *free cell* exists,
          indicates a location of at least one *free cell* of
          that *size* within the *data storage space;* and

        • upon receiving an instruction to store a *data item,*
          determining using said first *data array* an

15   appropriate *cell size* of a *free cell* for storing the
          data item, and using said second *data array* to   '
          derive the location within the *data storage space* of
          a location of *afree cell* of the determined *cell
          size.*

20   Preferably, the *data item* will be stored in the said
free cell, the said *cell* will be converted into a *live
cell,* and the first and second *data arrays* will be
updated to indicate that said *cell* is no longer *free.*

        Preferably, upon receiving an instruction to add a

25 new *free cell* of a certain *size,* the location of said
cell will be added to the appropriate *field* in the second
data array, said *cell*
will be converted to *a free cell,* and the *related fields*
of the first *data array* will be updated to indicate that

30 *a free cell* of said *size* exists.

        No limitation is implied on the ordering of the
operations in this method: operations may be ordered

sequentially or non-sequentially except where constrained
by logical dependencies. :

In a fourth aspect the invention provides a memory
management method for controlling one or more *data*
storage devices which support a *data storage space,* the
data storage space containing:

- one or more *free cells,* each said *cell* containing at
  least one pointer to the location in the *data*
  storage space of another said *cell;* and

- a *field* which stores the location within the data
  space of at least one of: (i) one of said *cells*
  which is at least as large as any other of said
  cells, or (ii) one of said *cells* which is at least
  as small as any other of said *cells;*

- said pointers being such that, starting from a *cell*
  of which the location is indicated in said *field,* '
  any other of said *cells* can be reached by moving
  successively from one *cell* to another according to
  the pointers in a path in which the *size* of the
  cells varies monotonically;

- the method including identifying *a free cell* of a
  certain *size* by starting from a first *free cell,* and
  moving successively from one *cell* to another
  according to the pointers until a *cell* of that *size*
  is reached.

Preferably, upon receiving an instruction to store a
data item of a certain *size,* an appropriate *free cell*
will be determined using the method described in the
fourth aspect and starting the search from the said
field, the *data item* will be stored in the said *free*
cell, the said free *cell* will be converted into a *live*
cell, and the *cell* will be detached from the linked- list
of free cells.

Preferably, upon receiving an instruction to add a new *free cell* of a certain *size,* said *cell* will be converted to *a free cell* and added to the linked-list *of free cells* in the appropriate position such that the
5   properties expressed in the fourth aspect are maintained (i.e. the pointers provide a path in which the *size* of the *cells* varies monotonically).

No limitation is implied on the ordering of the operations in this method: operations may be ordered
10   sequentially or non-sequentially except where constrained by logical dependencies.

Preferably, the first and fourth aspects of the invention will be combined so that, upon receiving an instruction to add a new *free cell* of a certain *size,* or
15   to allocate a *data item* of a certain *size,* the appropriate position for adding the new *free cell* to or , deleting an existing *free cell* from the linked-list *of free cells* may be determined by inspecting the first and second *data arrays* and said *data arrays* will be updated
20   accordingly to indicate either the presence of a new *free cell* or that an *existing free cell* has been deleted. This has the effect that both adding a new *free cell* and allocating an *existing free cell* can be achieved in O (logp(MAXFIELDS)) time.

25   In both the third and the fourth aspects of the invention, the memory management method preferably further includes a step of identifying at least one *cell* of a type other than a *free cell* (e.g. a *live cell* or a *zombie cell)* and converting it into *a free cell.*

30   Preferably, when a said *cell* is converted into a *free cell,* and if the *reference data* required *for a free cell* exceeds the available fields for *reference data* in the said *cell,* then *the fields* of the said *cell* that were used *for non-reference data* may be used to store the

reference data for the *free cell* after conversion.
Preferably also, when *a free cell* is converted to a *live
cell* as the result of allocation, those *fields* in the
free cell that were used for *reference data* may be used

5   for non- *reference data.*

         For this purpose the memory management method
preferably employs a data structure
according to the first or second aspect of the invention
in relation to *cells* of this other type.

10   In a fifth aspect, the invention proposes a memory
management method for controlling one more *data storage
devices* which support a *data storage space* containing
one or more *zombie cells,* each *zombie cell* containing at
least one pointer to the location in the *data storage*

15 *space* of another *zombie cell;*

   • the method employing *afield* which stores the location
     within the; data storage space of one of said *zombie
     cells* which is at least as large (or as small) as any
     other of said *zombie cells;*

20 • said pointers being such that, starting from a *zombie
     cell* of which the location is indicated in said *field,*
     any other of said *zombie cells* can be reached by
     moving successively from one *zombie cell* to another
     according to the pointers in a path in which the *size*

25   of the *zombie cells* varies monotonically;

   • the method including freeing *zombie cells* in an order
     defined by starting from a first *zombie cell,* and
     moving successively from one *zombie cell* to another
     according to the pointers.

30   Preferably, upon receiving an instruction to free a
zombie cell, an appropriate *zombie cell* will be
determined using the method described in the fifth aspect
and starting the search from the *said field,* the said

zombie cell will be converted into *a free cell,* and *said free cell* will be detached from the linked-list of *zombie cells.*

Preferably, upon receiving an instruction to add a
5   new *zombie cell* of a certain *size,* said *cell* will be converted to a *zombie cell* and added to the linked-list of *zombie cells* in the appropriate position such that the properties expressed in the fifth aspect are maintained (i.e. the pointers provide a path in which the *size* of
10   the *cells* varies monotonically).

No limitation is implied on the ordering of the operations in this method: operations may be ordered sequentially or non-sequentially except where constrained by logical dependencies.

15   Preferably, the first and fifth aspects of the invention will be combined so that, upon receiving an , instruction to add a new *zombie cell* of a certain *size,* or upon receiving an instruction to free a *zombie cell,* in the former case the appropriate position for adding
20   the
new *zombie cell* to the linked-list of *zombie cells* may be determined by inspecting the first and second *data arrays* and in both cases said *data arrays* will be updated accordingly to indicate respectively
25   either the presence of a new *zombie cell* or the deletion of a *zombie cell.* This has the effect that adding a new *zombie cell* or freeing the largest *zombie cell* can be achieved in O(logp(MAXFIELD)) time.

In a sixth aspect, the invention proposes that the
30   concepts described above are generalised by replacing cell size with any integer-valued property associated with a cell.

Specifically, in this a sixth aspect, the invention proposes a memory management method for controlling one

more *data storage devices* which support a *data storage space* containing one or more *cells,* each *cell* containing at least one pointer to the location in the *data storage space* of another *cell,* and each *cell* additionally associated with (e.g. containing) at least one integer quantity or property (which we call the *"sort value");*

- the method employing *afield* which stores the location within the data storage space of one of said *cells* which contains a *sort value* that is at least as large (or at least as small) as any other *sort value* of said *cells;*

- said pointers being such that, starting from a *cell* of which the location is indicated in said *field,* any other of said *cells* can be reached by moving successively from one *cell* to another according to the pointers in a path in which the *sort value* of the *cells* varies monotonically;

- the method including deleting *cells* in an order defined by starting from a first *cell* and moving successively from one *cell* to another according to the pointers.

Preferably, upon receiving an instruction to delete a *cell,* an appropriate *cell* will be determined using the method described in the sixth aspect and starting the search from the *said field,* and the said *cell* will be detached from the linked-list of *cells.*

Preferably, upon receiving an instruction to add a new *cell* of a certain *sort value,* said *cell* will be added to the linked-list of *cells* in the appropriate position such that the properties expressed in the sixth aspect are maintained (i.e. the pointers provide a path in which the *sort value* of the *cells* varies monotonically).

No limitation is implied on the ordering of the operations in this method: operations may be ordered

sequentially or non-sequentially except where constrained by logical dependencies.

Preferably, the first and sixth aspects of the invention will be combined so that, upon receiving an instruction to add a new *cell* of a certain *sort value,* or upon receiving an instruction to delete a *cell,* in the former case the appropriate position for adding the new *cell* to the linked-list of *cells* may be determined by inspecting the first and second *data arrays* and in both cases said *data arrays* will be updated accordingly to indicate respectively either the presence of a new *cell* or the deletion of a *cell.* This has the effect that adding a new *cell* or deleting the cell with the largest *sort value* can be achieved in $O(logp(MA.XFIELDS))$ time.

The above defined sixth aspect of the current invention thus provides a "priority queuing" mechanism for sorting *data items* of a certain type with respect to any integer quantity or property of said *data items* or of the *cells* that contain said *data items.* A "priority queue" is a data structure and associated methods which support the insertion of *data items* in any order and the selection (i.e. deletion) of *data items* in a predetermined order (for example, in order of the *size* of the *cells* that contain said *data items,* or in order of the value of the *data item,* which mayor may not be numerical). Thus, it is a mechanism for sorting *data items* into a certain order. Priority queues are important for a wide range of computational tasks: for example, the scheduling of processes by an operating system, the routing of prioritised data in a communications network (for example, the Internet), the servicing of prioritised requests by an Object Request Broker, and the processing of data from multiple sensors (such as found, for example, in aircraft, spacecraft and self-navigating

missiles). When said computational task has real-time
constraints, the time performance of the priority queue
is a critical element of the successful implementation of
that task.

5   Priority queues typically exhibit insertion and
deletion time overheads that either both scale
logarithmically with the number of *cells* in the queue, or
one scales linearly with the number of *cells* in the queue
and the other takes constant time. By contrast, the sixth
10   aspect of the current invention provides a priority
queuing mechanism that has deletion and insertion time
overheads that scale as follows:

- where MAXFIELDS and MINFIELDS limits are set,
  O(logp (MAXFIELDS - MINFIELDS));
15   - where no limit is set, and the data arrays are not
  dynamically re-balanced, O(logp(maximum *cell*   ,
  value));
- where no limits are set, and the data arrays are
  dynamically re-balanced, O(logp(number of
20   different cell values)).

This has the effect that insertion and selection may
be much faster than conventional priority queues in
situations where there are many duplicates and where the
values are clustered.

25   A further useful property of the current invention
is that it allows data items of equal value (in terms of
the criterion for selection) to be selected in order of
age (either oldest-first or youngest-first). Most
implementations of priority queues do not provide this
30   facility. This has the effect that stronger guarantees
can be provided about the behaviour of computer software
that uses the current invention as the basis for priority
queuing.

In further aspects the invention provides memory
management methods (fully combinable with the methods
according to any of the third to sixth aspects of the
invention) in which a data structure according to the
5   first or second aspect of the invention is updated by the
insertion into it of a cell of the certain type, and the
first and second tables and any pointers are updated
accordingly. Preferably, this insertion preserves any
information about the cells encoded by the pointers.

10

In further aspects, the invention provides a *data
storage device* which is arranged to store a data
structure according to the first and/or second aspects of
the invention, and/or to perform any of the methods
15   according to the invention (including means for
performing each respective step of the methods). The
invention further provides a computer system for
performing a *computational task* and including such a *data
storage device*.

20   Any of the above aspects may be fully combined, as
demonstrated in two examples of the invention that will
now be described in detail for the purpose of example
only.

25   Detailed Description of two examples of the invention

The first example describes a priority queue that does
not limit the sizes of the first and second data arrays
(i.e. there is no MAXFIELDS limit). The second example
30   describes a memory allocation system where the sizes of
the first and second arrays are limited by MAXFIELDS.

Priority Queue

In this example the first and second *data arrays* of
the first aspect of the current invention are merged for
ease of implementation. The result is a single *data array*
that contains both those *fields* determining the existence

5   of *cells* of a certain value and those *fields* determining
the location of said *cells* if they exist.

The *data array* comprises a hierarchy of memory
"blocks". Each block consists of a bitmap (in this
example, an unsigned 32-bit integer) and one or more

10   *fields* held in contiguous locations (so that standard
array indexing may be used to locate *afield* in constant
time), with *each field* containing a pointer (in this
example each block contains 32 pointers). At the lowest
level, the pointers give the locations of cells: at all

15   other levels the pointers hold the addresses of blocks in
the next lower level. Additionally the *data array*   ,
contains *afield* holding a pointer to the top layer of the
hierarchy, and *afield* that stores the current number of
layers in the hierarchy. The bitmaps provide the

20   implementation of the first *data array* of the first
aspect of the current invention and the pointers provide
the implementation of the second *data array* of the first
aspect of the current invention.

Initially there exists a *sentinel cell* that had a

25   known fixed address and is never subsequently moved or
deleted. Each *cell* contains reference data including (i)
the value of said cell to be used for ordering subsequent
selection, and (ii) a pointer to the "next" *cell* of lower
or equal value (if such a *cell* does not exist, this *field*

30   is empty), and (iii) a pointer to the "previous" *cell* of
higher or equal value (if such a *cell* does not exist, the
pointer gives the location of the *sentinel cell*). Thus,
the *cells* are arranged as a double-linked list. The
double- linked list is at all times maintained in

monotonic order of *cell* value. This arrangement provides
the implementation of the second aspect of the current
invention.

To provide an implementation of a priority queue,
5   the first and second aspects of the current invention are
combined as follows. The pointers of the lowest level of
the *data array* point to *cells* in this list -there is a
single pointer from the *data array* for each different
value that is currently being used for sequencing in the
10   priority queue. For a certain value, if duplicate *cells*
with that value exist, they are grouped next to each
other in the double-linked
list and the pointer from the *data array* for that value
gives the location of the cell which is linked to the
15   *cell* with the next highest value (or to the *sentinel*
cell, if no higher valued *cell* exists). This is
illustrated in Figure 2.

The priority queue is initialised as follows:

- The *sentinel cell* is empty and there are no other
20   *cells;*

- The *data array* has one block containing *32 fields*
   and a bitmap, each *field* being NULL and each bit in
   the bitmap being 0;

- The pointer to the top layer of the data array (we
25   call this variable "top") contains the location of
   the first *field* (that which points to the *sentinel*
   cell);

- The variable which gives the number of layers in the
   data array (we call this variable "layers") is set
30   to the value 1;

- Upon receipt of an instruction to insert a new cell
   into the priority queue, the following actions
   occur:

- The *reference data* of said *cell* is inspected to
  determine the value that will be used for subsequent
  selection —we call this value "x", We call said *cell*
  "this cell" and its previous and next pointers are
  called "previous" and "next",

  - First we determine whether the *data array* has
    sufficient layers to support the insertion, If
    not, we create new layers:

```
while (x >= (32^layers)){
    ptr = new (block) ;        /*initialised empty */
        ptr[0] = top;
        top = ptr;
        layers = layers + 1;
        }
```

  - Next, the following code performs an O(log N)
    search of the *data array,* where N is the largest
    value of any cell in the priority queue. If blocks
    are missing from the hierarchy, they are created.

```
        a = top;
                b = layers;
While (b > 1)
        c = x div (32^(b-1)) -(32^(b-1)) *(x div
(32^b));
        if (a[c] == NULL) {

        ptr = new (block) ; /*initialised with
NULLs */
        a[c] = ptr;
        }
        a = a [c] ;
        b = b -1;
        }
        result = a[x mod 32] ;
```

- If the result is not empty (the pointer is not NULL) then simply de-reference the pointer to find the cell and link the new cell into the double-linked

5   list of cells (the previous "result" pointer should be updated to point to the new cell). Otherwise update the "result" pointer to point to the new cell and undertake a search of the bitmaps in the *data array* to determine the previous and next cells in

10   the double linked-list to which the new cell should be linked, and to set the appropriate bits in the related bitmaps to indicate the existence of the new cell (this procedure is explained in the second example, below, which considers a memory allocation

15   system).

Upon receipt of an instruction to select (delete) the largest valued cell, this is achieved in O(log N) time (as with insertion) by (i) following the sentinel cell pointer to find the

20   largest cell, (ii) detaching the largest cell by adjusting the sentinel pointer and the "previous" pointer in the next cell as necessary, (iii) traversing the data array from the top layer to the lowest layer to update the appropriate bits in the related bitmaps, and (iv)

25   having reached the lowest level, updating the pointer in the data array (either to set it to NULL or to set it to point to the next cell of the same value as the cell just deleted).

30   Memory Management Method

This example of the invention is a memory allocation system where the sizes of the first and second arrays are limited by MAXFIELDS.

The memory allocation system provides best-fit memory allocation with allocation delays that are independent of the size of the memory.

The memory is divided into blocks. Each block comprises both an administrative data area and a user data area. If the block is not being used by the program (known as a "free" block), the user data held in the user data area is irrelevant and the user data area may therefore be used to hold additional administrative data. If the block is being used by the program (known as a "live" block) then the user data area may not be used for administrative data.

The administrative data for all blocks is the same and comprises:

(i) the size (the number of bytes) of the user data area

(ii) whether the block is free or live

(iii) whether the previous block (the block occupying an adjoining position in memory at a lower memory address) is free or live

(iv) the size (the number of bytes) of the user data area of the previous block

The sizes are expressed as numbers of bytes. The size of the user data area for a live block may not exceed MAXFIELDS bytes. The size of a free block may, however, exceed MAXFIELD bytes (as described below).

When the system starts, the memory contains amongst other things a single very large free block that represents the entire memory that the program may use. The size of this

block (to which we give the name "wilderness block") will be bigger than MAXFIELDS. Also, during the course of the program run, adjacent free blocks may be coalesced, and may thereby create a free block whose size is bigger than
5 MAXFIELDS (we also give these the name "wilderness blocks").

This example describes a memory management system where MAXFIELDS is less than 32,768 and therefore both the availability (free or live) and the size of a block can
10 be expressed with a 16 bit signed integer. The sign bit indicates whether the block is free (negative) or live (positive) and the magnitude indicates the size. An integer value of -32,768 indicates a free block that is bigger than 32,767 bytes and the actual size of the free
15 block is stored in the data area of that block (see below).

Each free block contains, in addition to the required administrative data of two 16-bit integers (one for size
20 and availability of this block, and one for the size and availability of the previous block), two pointers to other free blocks. The first such pointer (to which we give the name "leftp") gives the memory address of the free block that is next highest in a double-linked chain
25 of free blocks. The second such pointer (to which we give the name "rightp") gives the memory address of the free block that is next lowest in the double-linked chain of free blocks. These two pointers are held at the start of the user data area of a free block. Each pointer
30 requires 4 bytes and so the user data area of a block in this system must always be at least 8 bytes. If the program requests a block of less than 8 bytes user data area, it will be provided with a block of 8 bytes user data area (a request for a negative size is treated as an

error). This introduces a small amount of internal
fragmentation. Wilderness blocks hold the actual size of
the user data area in the user data area itself,
immediately after the "rightp" pointer. A copy of the
size is also held at the very end of the user data area
(to assist coalescing of free blocks).

In summary, the three types of block have the following
layout:

1. live block

    administrative data area:
        2 bytes - size and availability of this block -
must be positive
        2 bytes - size and availability of previous block
    user data area:
        N bytes - user data area, where N >= 8

2. free block (not wilderness)

    administrative data area:
        2 bytes - size and availability of this block -
must be negative
        2 bytes - size and availability of previous block
    user data area:
        4 bytes - "leftp" - memory address of next block
in free list
        4 bytes - "rightp" - memory address of previous
block in free list
        N bytes - user data area, where N >= 0

3. wilderness block (free)

administrative data area:

       2 bytes - size and availability of this block -
must be -32,768

       2 bytes - size and availability of previous block

     user data area:

       4 bytes - "leftp" - memory address of next block
in free list

       4 bytes - "rightp" - memory address of previous
block in free list

       4 bytes - actual size of user data area

       N bytes - remainder of user data area - where N >=
32,752

       4 bytes - actual size of user data area

At initialisation, the memory comprises three blocks:

- a "sentinel" block (a live block used by the system) at
  the lowest address

- a "max" block (a live block used by the system)

- a "wilderness" block (available to the program)

The system also maintains:

(i) an array (to which we give the name "pgarray") of
MAXFIELDS+1 pointers to free blocks that are not
wilderness blocks. For administrative purposes the first
pointer in this array (corresponding to a free block of
size zero, which cannot exist) is initialised to contain
the memory address of the "sentinel" block.

(ii) a pointer to the "max" block (which may, but need
not, be held in the array described in (i), since entries
1-7 of that array will be unused)

(iii)a 3-level hierarchy of bitmaps containing
information about the data held in the array described in
(i). There need only be three levels if MAXFIELDS is

less than 32,768. The first level of bitmaps is an array
(to which we give the name bitmap0) of 1024 unsigned
integers. This will support up to 32768 entries in
pgarray. The second level of bitmaps is an array (to

5   which we give the name bitmap1) of 32 unsigned integers;
this will support up to 1024 entries in bitmap0. The
final level of bitmaps is a single unsigned integer to
which we give the name bitmap2; this supports up to 32
entries in bitmap1.

10 (iv) a linked-list of free blocks in size order, with the
lowest block in this linked list being the "sentinel"
block and the highest block in this linked list being the
"max" block. Note that the "sentinel" and "max" blocks
are both live blocks, yet they are the two ends of the

15 double-linked free-list. It is important that they are
live, not free (unlike all other blocks on the free list)
because they must never be allocated and they must never
be coalesced with other blocks.

20 If this memory management system is to be used for a
program that allocates only individual bytes of data,
then fragmentation may be minimised by permitting the
system to allocate blocks starting at any address in
memory. However, if the program allocates larger units of

25 data such as integers then additional internal
fragmentation will occur because the system will be
required only to allocate blocks where the user data
starts at a memory address that is a multiple of 4
(alternatively, 8-byte or 16-byte alignment may be

30 required); to achieve this, it is necessary to ensure
that (i) the initial wilderness block has a user data
area that starts at a suitably aligned memory address and
(ii) all subsequent requests for block allocation are
rounded up to the nearest size such that the neighbouring

free block (at higher address) has a user data area that is correctly aligned.

The method proceeds as follows:

5

1. an initialisation method ensures that:

- the bitmaps firstly have all bits set to zero
- the bitmaps then have the first bit at each of the three levels set to 1 (this is because pgarray will
10 be initialised with pgarray[0] containing the address of the sentinel block)
- pgarray[0] is given the memory address of the sentinel block, which is defined to be at the lowest end of memory
15 - the sentinel block followed by the max block are allocated at the lowest end of memory alternatively,' the max block could be placed at the upper end of memory). The "previous size and availability" data held in the max block are set to the appropriate data
20 for the sentinel block.
- two variables with the names "sentinelp" and "maxp" are set to contain the addresses of the sentinel block and the max block
- a wilderness block is created in memory immediately
25 after the max block, occupying all of the rest of the available memory. Because it is a wilderness block, pgarray does not contain an entry for this block. However, the block must be linked into the double-linked free list. The "previous size and
30 availability" data for the wilderness block are set to the appropriate values for the max block. The "size and availability" data for the wilderness block are set to -32,768.

- the leftp for the sentinel block and the rightp for the max block are both set to the memory address of the wilderness block
- the rightp of the sentinel block and the leftp of the max block are both set to zero
- the leftp of the wilderness block is set to the address of the max block, and the rightp of the wilderness block is set to the address of the sentinel block.
- the size of the wilderness block is stored in the wilderness bloc just after the rightp and a copy is stored at the end of the user data area of the wilderness block

2. A main method receives memory requests from the program and allocates memory appropriately. Three different requests may be received: "malloc", "realloc" and "free".

The first is a request for the allocation of new memory of a given size; the main method finds the free block that is closest to the given size and splits that block into two parts – the lower part has a user data area that is the size required by the program (subject to alignment constraints as indicated above) and its memory address is returned to the program, whilst the upper part is retained as a free block for future use. The third request is that a live block at a given address should no longer be considered as live – it is therefore linked into the free list so that it may be available for future use. The second request is that a live block should be resized (either to a greater or smaller size), with the data in the current block being retained – this may be achieved by growing or shrinking the existing live block, or by allocating a new block, copying the data, and

freeing the existing block. For each request, the main
method passes control to a subsidiary method to deal with
the request. The three subsidiary methods are given the
names "ccmalloc", "ccrealloc" and "ccfree". These three
5   methods are described in detail below.

3. The method "ccmalloc"

This method is provided with the size of user data area
10   that is requested. The method allocates the best-fit free
block, unlinks that free block from the double-linked
chain of free blocks, resets the pgarray and bitmaps a
necessary, updates the administrative data for the block
to show that it is now a live block, and returns the
15   memory address of the block. If the selected free block
is larger than the requested size, the remainder of the ,
block may be divided from the requested part and the
remainder is then treated as a new free block and linked
into the double-linked chain of free blocks and the
20   pgarray and bitmaps are updated as appropriate.

When ccmalloc is called for the first time, there will be
no entries in pgarray apart from the pgarray[0] entry
that points to the sentinl block. The leftp pointer of
25   the sentinel block points to the wilderness block, which
can then be split to satisfy the request.

The method is described in detail as follows:

30 • first the requested size is set to 8 if it is less than
      8
    • if the size is greater than MAXFIELDS, the allocator
      terminates with an error

- if pgarray[size] is not empty then a free block of the requested size is available; this block is changed from being a free block to a live block by passing the memory address of the identified block as an argument

5 (together with the requested size) to a method with the name "ccdelete" (this method is described below). The address of the block is then returned as the result of this method "ccmalloc"

- if pgarray[size] is empty then no free block of the

10 requested size exists, and so a search must be made for the next biggest free block.

- control is passed to the method "search"; with the requested size passed as an argument. The "search" method is described below. It returns the index in the

15 pgarray corresponding to the next smallest free block

- the leftp of the next smallest free block contains the memory address of the next biggest free block. This is the block that will be used to satisfy the request. We name this the "target" block

20 - if the target block is a wilderness block then we chose a new target block which is the wilderness block whose address is held in the rightp of the max block. This switch (together with appropriate actions when free blocks are coalesced) ensures that those wilderness

25 blocks that are created by coalescing are chosen to fulfil allocation requests in preference to the wilderness block at the end of memory - this in turn reduces the overall memory requirements of the program.

- the amount of memory to be taken from the

30 wilderness block will be the requested size (the administrative data at the start of the wilderness block can be reused for the newly-allocated block

- the size of the remaining block will be the current size of the target block less the requested size less the size of the administrative data for the remaining block - we give this value the name "j"

- the administrative data for the remaining block is set such that the "previous availability and size" is positive and is the requested size

- the administrative data for the newly allocated block is set such that "my size an availability" is positive and is the requested size

- if the size of the remaining block is less than 32,768 then the administrative data for the remaining block is set such that "my size and availability" is negative and has the value given by "j"

- and the memory address of the remaining block is passed as an argument to the method "ccfree" if the size of the remaining block is greater than 32,767 then the administrative data for the remaining block is set such that "my size and availability" is the value -32,768 and the real size given by "j" is stored just after the rightp for the remaining block and a copy of "j" is stored at the end of the user data area of the remaining block; the leftp for the remaining block is set to be a copy of the leftp for the target block and the rightp for the remaining block is set to be a copy of the rightp for the target block.Next the leftp of the block whose memory address is contained in the rightp of the target block is set to be the memory address of the remaining block and the rightp of the block whose

memory address is held in the leftp of the target

block is set to be the memory address of the

remaining block the memory address of the newly

allocated block is returned as the result of the

5   "ccmalloc" method

- if the target block is not a wilderness block then the

  address of the target block is passed (together with

  the requested size) as an argument to the method

  "ccdelete"

10 - the memory address of the newly allocated block is

     returned as the result of the "ccmalloc" method

4. The method "ccrealloc"

This method resizes an already-allocated (live) block.

15 It takes two arguments: the memory address of the block

to be resized, and the new size that the block should

have. It returns the address of the resultant block.

- if the memory address is 0, there is no block to

20   resize and so ccrealloc simply passes control to

        the method ccmalloc(size) and returns whatever

        memory address is returned by ccmalloc

- if the size is zero, the live block must be freed

        and so ccrealloc simply passes control to the method

25   ccfree(address) and then returns a memory address of

        zero.

- the size of the block at the given memory address

        (the "current" block) is inspected (give it the

        name "current size") and is used to calculate the

30   memory address of the next highest adjacent block in

        memory.

- the size of the next highest adjacent block in memory is inspected (we give this the name "nextsize")

- if nextsize indicates that the next block is a wilderness block then the true next size is read from the user data area of the wilderness block

- nextsize is set to its absolute value (the sign bit is ignored)

- if current size is the same as size then no work need be done - the ccrealloc method terminates and returns the original memory address as its result

- if the requested size is greater than the current size AND the current block is not the highest block in memory AND the next highest block in memory is free AND the requested size is less than or equal to MAXFIELDS AND (nextsize + current size + the size of the administration data area) is greater than the requested size AND (nextsize + current size) is greater than (the requested size + the minimum size of the user data area (in this example 8)) THEN perform the following actions:

  - calculate (requested size - current size - size of administration data area) and give it the name "growsize"

  - if growsize is negative, set growsize to zero

  - calculate (current size + growsize + size of administration data area) and give it the name "newsize"

  - pass control to the method "ccdelete", passing it three arguments: (i) the memory address of the next highest block, (ii) growsize and (iii) the Boolean value TRUE (which indicates that a value

for growsize of less than 8 should not be treated
as an error nor be rounded up to 8)

- if newsize is greater than MAXFIELDS then
  terminate with an error message

- update the administrative data area for the
  current block such that "my size and availability"
  is set to be newsize (which should be positive,
  because this remains a live block)

- update the administrative data area of the newly-
  formed next highest block in memory such that its
  "previous size and availability" is set to be
  newsize (which should be positive)

- terminate this method and return the memory
  address of the current block

- if the previous test failed, then it is not possible
  (or not necessary)   simply to grow the current block'
  into the next highest block, so either the block
  should be shrunk or a new block should be allocated of
  the required size and the live data in the user data
  area of the current block should be copied to the new
  location.   In fact, the latter approach subsumes the
  former in terms of correctness (if not in terms of
  efficiency!) and so to simplify this example we shall
  always allocate and copy even in situations where the
  current block could have been shrunk:

  - pass control to the method ccmalloc, passing it
    the requested size as an argument. The memory
    address returned by ccmalloc is given the name
    "cp2"

  - loop N times, where N is the smaller of the
    requested size and the current size, each time
    around the loop copying a byte of user data from

the current block to the block at memory address
cp2

- pass control to the method ccfree, passing the
  memory address of the current block as an argument
  terminate this method and return as a result the
  memory address cp2

5. The method "ccfree"

The method ccfree frees a block that is not currently
linked into the double-linked chain of free blocks; it
does this by linking the block into the double-linked
chain of free blocks and updating the pgarray and bitmaps
accordingly. It takes as an argument the memory address
of a block to be freed. it does not return any result.

- the memory address passed to ccfree is the address of a
  block to which we shall give the name "current block"
- the "previous size and availability" for the current
  block is inspected and stored in a local variable with
  the name "prevsize" and the "my size and availability"
  of the current block is inspected and stored in a
  local variable with the name "thissize"
- if thissize is -32,768 then read the actual size of the
  block from the memory location just after rightp,
  store this actual value in a local variable with the
  name "size", and calculate the memory address of the
  next highest block by adding the value of size (plus
  the size of the administration data area) to the memory
  address of the current block and store this value in a
  local variable with the name "nextblock"
- if thissize is not -32,768 then calculate the memory
  address of the next highest block by adding the value
  of thissize (plus the size of the administration data

area) to the memory address of the current block and
store this value in a local variable with the name
nextblock"; also set the value of size to be the same
as the value of thissize

5   • if prevsize is -32768 then read the actual size of the
previous block from the 4 bytes just before the
administration data area of the current block (these
are the top 4 bytes of the user data area of the
previous block) and calculate the memory address of

10   the previous block by subtracting this actual size
(together with the size of the administration data
area) from the memory address of the current block and
store this value in a local variable with the name
"prevblock"

15   • if prevsize is negative but not -32,768, calculate the
memory address of the previous block by adding
prevsize to the memory address of the current block
and then subtracting the size of the administration
data area and store this value in a local variable with

20   the name "prevblock"

    • otherwise, prevsize must be positive, so calculate the
memory address of the previous block by subtracting
prevsize (plus the size of the administration data
area) from the memory address for the current block and

25   store this value in a local variable with the name
"prevblock"

    • inspect the "my size and availability" of the next
highest block and store it in a local variable with
the name "nextsize"

30

    • if (thissize is negative) AND (thissize is not -32,768)
then terminate the method with an error message
(because the only free blocks that might not be already
attached to the free list are wilderness blocks)

The ccfree method then determines whether the current block can be coalesced with the next block in memory and/or the previous block in memory (one or both are already free). First the next block is considered, and then the previous block.

- if nextsize is negative then merge this block with the next block by executing the following instructions:
  - if nextsize is NOT -32,768 then it is not a wilderness block and so execute the following instructions:
    - if (size - nextsize + size of administration data area) is greater than MAXFIELDS then the result of coalescing will be a wilderness block so execute the following instructions:
      - pass control to the ccdelete method, passing three arguments: (i) nextblock, (ii) nextsize, and (iii) the Boolean value FALSE
      - calculate the memory address of the block after the next block as (nextblock - nextsize + the size of the administration data area) and store it in a local variable with the name "nextnextblock"
      - update the administration data area of the current block such that "my size and availability" is set to -32,768
      - update the memory address just above rightp for the current block such that it contains the value (size - nextsize + size of administration data area)
      - update the memory address 4 bytes before the administrative data area of the block after the

next block, such that it contains the value
(size - nextsize + size of administration data
area)

- if the block after the next block exists (i.e.
if it is a valid memory address) then update
the administration data area of that block such
that "previous size and availability" is set to
be -32,768

- if (size - nextsize + size of administration data
area) is NOT greater than MAXFIELDS then the result
of coalescing will not be a wilderness block so
execute the following instructions:

  - update size with new value (size - nextsize +
  size of administration data area)

  - update the administration data area of the
  current block with the value -size (NB it must
  be negative)

  - calculate the memory address of the block after
  the next block as (nextblock - nextsize + the
  size of the administration data area) and store
  it in a local variable with the name
  "nextnextblock"

  - pass control to the method ccdelete, passing it
  three arguments: (i) nextblock, (ii) the value
  -size, and (iii) the Boolean value FALSE

  - if the block after the next block exists (i.e.
  if it is a valid memory address) then update
  the administration data area of that block
  such that "previous size and availability" is
  set to be -size

- if nextsize IS -32,768 then it is a wilderness block
and so execute the following instructions:

- get the actual size of the next block and store
  it in a local variable to which we give the
  name "bignextsize"

- pass control to the method ccdelete with three
  arguments: (i) nextblock, (ii)
  bignextsize,(iii) the Boolean value FALSE

- update the administration data area of the
  current block such that "my size and
  availability" is set to -32,768

- calculate the new size which is (size +
  bignextsize + the size of the administration
  data area) and update the local variable size
  to contain this new value

- store the new value size in the current block
  at the memory address just after rightp

- store the new value size in the memory address
  obtained by adding size to the memory address
  of the current block minus 4 bytes

- calculate the value (bignextsize + nextblock +
  the size of the administration data area)and
  store this in the local variable with the name
  "nextnextblock"

- update the local variable nextblock to have the value
  stored in the variable nextnextblock

The ccfree method next considers the possibility of
merging with the previous block.

- if prevsize is negative then merge this block with the
  previous block by executing the following
  instructions:

- if prevsize is NOT -32,768 then the previous block is not a wilderness block and the following instructions should be executed:

- update the variable size so that it has a new value given by the calculation (size - prevsize + size of administration data area)

- pass control to the method ccdelete, passing the three arguments (i) prevblock, (ii) - prevsize, and (iii) the Boolean value FALSE

- set the current block to be the memory address of prevblock

- if size is greater than MAXFIELDS then the coalescing has created a wilderness block and the following instructions should be executed:

- update the administration data area of the current block such that "my size and availability" is set to -32,768

- save the value size at the memory address just after rightp in the user data area of the current block

- save the value size at the memory address given by the calculation (prevblock + size - 4)

- if nextblock is a valid memory address, update the administration data area of nextblock such that "previous size and availability" is set to -32,768

- if size is not greater than MAXFIELDS then update the administration data area of the current block such that "my size and availability" is set to the value -size; then (if nextblock is a valid memory address) update the administration data area of the next block such that "previous size and availability" is set to -size

- if prevsize IS -32,768 then the previous block is a wilderness block, so execute the following instructions:

  - read the data held in the memory address 4 bytes before the administration data area forthe current block and store this in a local variable with the name "bigprevsize"

  - store in the memory address just after rightp for the previous block the value given by the calculation (size + size of administration data area + bigprevsize)

  - store the value given by the calculation (size + size of administration data area + bigprevsize) into the memory address given by the calculation (prevblock + size + size of administration data area + bigprevsize - 4).

  - if nextblock is a valid memory address, update the administration data area of nextblock such that "previous size and availability" is set - 32,768

  - pass control to the method ccdelete with three arguments as follows: (i) prevblock, (ii) size + bigprevsize + size of administration data area, and (iii) the Boolean value FALSE

  - update the current block so that it refers to the memory address stored in prevblock

The ccfree method has at this stage finished all coalescing and subsequently adds the current block (possibly now coalesced) into the free list and pgarray.

- inspect the administrative data area of the current block and store the value of "my size and availability" into a local variable with the name "thissize"

5 - if thissize is -32,768 then the current block is a wilderness block and the following instructions should be executed:

- - the current block is inserted at the top of the double-linked chain of free blocks, pointed to
10   directly by the max block. This is achieved by updating the leftp of the current block to contain the memory address of the max block, by updating the rightp of the current block to contain a copy of the value currently stored in the rightp of the
15   max block, by updating the leftp of the block whose memory address is stored in the rightp of the max block so that it contains the memory address of the current block, and by updating the rightp of the max block so that it contains the
20   memory address of the current block

- - the ccfree method then terminates and control passes to the method that called ccfree.

- if thissize is NOT -32,768 then the current block is
25 not a wilderness block and the following instructions should be executed:

- - set thissize to be the absolute value of thissize (i.e. make it positive if it is negative)

- - if (thissize is less than or equal to MAXFIELDS)
30   AND (pgarray[thissize] is not empty) then link the current block into the double-linked list of free blocks just above the block whose memory address is held in

pgarray[thissize], and make pgarray[thissize]
hold the memory address of the current block.

This is done by executing the following
instructions:

5
- store in the leftp of the current block the
  memory address currently stored in the leftp
  of the block whose memory address is stored
  in pgarray[thissize]

- store in the leftp of the block whose memory
10
  address is held in pgarray[thissize] the
  memory address of the current block

- find the block whose memory address is now
  stored in the leftp of the
  current block and in the rightp of that block
15
  store the memory address
  of the current block

- store in the rightp of the current block the
  memory address that is held in
  pgarray[thissize]

20
- Note that we do NOT make pgarray[thissize]
  point to the current block yet - this is done
  later

- if either (thissize is greater than MAXFIELDS) OR
  (pgarray[thissize] is empty)then it will be
25
  necessary to search for the correct place in the
  double-linked chain of free blocks in which to
  insert the current block. This is done by
  executing the following instructions:

  - pass control to the method
30
    "set_and_search_bitmap" with one argument
    passed, which is thissize; this method will
    return a number which is then stored in a
    local variable with the name "i"

- update the rightp for the current block so that it contains the memory address held in pgarray[i]

- update the leftp for the current block so that it contains the data in the leftp of the block whose memory address is held in pgaray[i]

- update the leftp of the block whose memory address is held in pgarray[i]   so that it contains the memory address of the current block

- update the rightp of the block whose memory address is held in the leftp of the current block so that it contains the memory address of the current block

- Next update the administration data area of the current block so that "my size and availability" is -thissize

- If the next block is a valid memory address, update its administration data area   such that "previous size and availability" is -thissize

- If thissize is less than or equal to MAXFIELDS, update the contents of pgarray[thissize] so that it contains the memory address of the current block

6. the subsidiary method "ccdelete"

The method ccdelete is used internally (by ccmalloc and ccfree) - it deletes (unlinks) a specified block from the double-linked chain of free blocks and removes any reference to that block from either the pgarray or the bitmaps. It will also manage the splitting of such a

deleted block if required, with the lower sub-sblock
remaining deleted and the upper sub-block being
reinserted back into the double-linked list of free
blocks (and pgarray and bitmaps updated accordingly).

5   The method takes three arguments (i) the memory address
of a block to be deleted, which will be stored in a local
variable with the name "cp" (ii) the size of the block to
be deleted, which will be stored in a local variable with
the name "size" (if it is less than the actual size of

10 the block, this indicates that a split is requested), and
(iii) a Boolean value to indicate whether it is OK to
split just a very small (< 8 bytes) block from the start
of the given free block

- this will be stored in a local variable with the name

15   "small_is_ok".

- inspect the administration data area of the block whose
  memory address is held in cp (we give this the name
  "current block") and store the "my size and
  availability" data into a new local variable with the

20   name "mysize"

- if mysize is negative and mysize is not -32,768 then
  update mysize with the value -mysize

- if size is negative than update size with the value -
  size

25 - read the data held in rightp of the current block and
  save it in a new local variable with the name
  "rightblock"

- store the value held in rightblock into the rightp of
  the block whose memory address is held in the leftp

30   of the current block

- read the value stored in the leftp of the current block
  and store it into the leftp of the block whose memory
  address is held in rightblock

- if (mysize is not -32,78) AND (pgarray[mysize] is equal to cp) then execute the following instructions:
    - if the "my size and availability" data in the administration data area of the block whose address is stored in rightblock is equal to - mysize then update pgarray[mysize] so thatit holds the memory address that is currently stored in rightblock
    - else update pgarray[mysize] to contain an empty address (i.e. the value zero) and then pass control to the method "clear_bitmap", with mysize as an argument (and, when the method clear_bitmap returns, continue executing the next instruction for this method)
- if mysize is not -32,768 then (i) update the administration data area of the current block such that "my size and availability" is set to the value mysize, and (ii) if the next highest block (given by the memory address cp + mysize + size of adminstration data area) is a valid memory address, update the "previous size and availability" data of the administration data area of the next block to be the value mysize
    - if (size is less than 8) AND (small_is_ok is FALSE) then set size to be 8
    - if mysize is -32,768 then read the actual size from the memory address just after rightp in the user data area of the current block and store it in a new local variable with name "bigmysize"
- else store the value currently held in mysize into a new local variable with name "bigmysize"

- if (bigmysize - size - the size of the administration data area) is less than 8 then update size to be the value held in bigmysize
- if size is bigger than bigmysize then terminate this method with an error message
- if bigmysize is bigger than size then execute the following instructions:
  - calculate (cp + size) and store the result in a new local variable with the name "cp2"
  - if ((bigmysize - size - the size of the administration data area) is greater than MAXFIELDS) then execute the following instructions:
    - update the administration data area of the block whose memory address is held in cp2 wuch that "my size and availability" is set to the value -32,768
    - if (cp + bigmysize + size of the administration data area) is a valid memory address then update the administration data area of the block at that address such that "previous size and availability" is set to -32,768
    - store the value (bigmysize - size - the size of the administration data area) into the memory location given by (cp2 + 8 + size of administration data area)
    - store the value (bigmysize - size - the size of the administration data area) into the memory location given by (cp2 + bigmysize - 4)
  - else execute the following instructions:

- update the administration data area of the
block whose memory address is held in cp2
wuch that "my size and availability" is set
to the value (bigmysize - size - size of
administration data area)

- if (cp + bigmysize + size of the
administration data area) is a valid memory
address then update the administration data
area of the block at that address such that
"previous size and availability" is set to
(bigmysize - size - size of administration
data area)

- update the administration data area of the
block whose address is stored in cp2 such
that "previous size and availability" is set
to the value held in size

- update the administration darta area of the
current block such that "my size and
availability" is set to the value held in
size

- pass control to the method "ccfree", with the
memory address currently held in cp2 as an
argument

- else if (cp + bigmysize + size of administration data
area) is a valid memory address then execute the
following instructions:

- calculate (cp + bigmysize + size of data
administration area) and store the result in
a new local variable with name "nextblock"

- if bigmysize is greater than MAXFIELDS then
update the administration data area
ofnextblock such that "previous size and
availability" is set to -32,768

- else update the administration data area of
  nextblock such that "previous size and
  availability" is set to bigmysize

7. The subsidiary method "search"

This method takes a single argument "size" and searches
the bitmaps to find the highest set bit that is less than
the bit representing the pgarray index "size". It
returns the appropriate index.

- if size is greater than MAXFIELDS set size to be
  MAXFIELDS
- calculate the integer result of dividing size by 32 and
  store this value in a local variable with the name
  "bitindex"
- calculate the remainder after dividing size by 32 and
  store this value in a local variable with the name
  "bitremainder"
- if bitremainder is zero execute the following
  instructions:
  - if bitindex is zero then terminate this method and
    return the value zero
  - if bitindex is not zero then update bitindex with
    the result of subtracting one from the existing
    value of bitindex and then update bitremainder
    so that it contains the value 32
- create a 32 bit integer with only the top bit set and
  all other bits being zero (i.e. 0x80000000) and store
  it in a local variable with name "msb"
- calculate the result of right-shifting msb
  (bitremainder-1) times and store the result in a local

variable with the name "mask". The value stored in msb
should NOT be altered.

- store the value (bitremainder-1) in a local variable
  with the name "i"

5 - loop until i is no longer positive, executing the
  following instructions each time around the loop:

  - if the result of the bitwise AND of
    bitmap0[bitindex] with mask is not zero then
    terminate this method and return the value given

10   by the calculation ((bitindex multiplied by 32) +
    i)

  - update mask to be the result of the existing value
    of mask left-shifted once

  - update i to contain (i - 1)

15

- calculate the integer result of dividing (bitindex - 1)
  by 32 and store this value in a local variable with
  the name "j"

- calculate the remainder after dividing (bitindex - 1)

20 by 32 and store this value in a local variable with
  the name "searchtop"

- update mask to contain the result of right-shifting msb
  searchtop times (the value held in msb should not be
  altered)

25 - set a local variable "foundit" to be the Boolean value
  FALSE

- update i to contain the value currently held in
  searchtop

- loop until i is no longer positive, executing the

30 following instructions each time around the loop:

  - if the result of the bitwise AND of bitmap1[j]
    with mask is not zero then update foundit to
    contain the Boolean value TRUE and store the

value   currently held in j into a new local
variable with name "bitindex2" and store the
value currently held in i into a new local
variable with the name bitindex1 and then break
out of this loop and continue with the remaining
instructions for this method

- update mask to be the result of the existing
  value of mask left-shifted once

- update i to contain (i - 1)

- if foundit contains the Boolean value FALSE then
execute the following instructions:

  - update mask to contain the result of right-
    shifting msb a number of times that is   given
    by the remainder after dividing (j-1) by 32

  - update i to contain the remainder after dividing
    (j-1) by 32

  - loop until i is no longer positive, executing the
    following instructions each time around the loop:

    - if the result of the bitwise AND of
      bitmap2[0] with mask is not zero, update
      foundit to contain the Boolean value TRUE and
      update bitindex2 to contain the value
      currently held in i and then break out of
      this loop and continue with the remaining
      instructions for this method

    - update mask to be the result of the existing
      value of mask left-shifted once

    - update i to contain (i - 1)

  - if foundit contains the Boolean value FALSE then
    there has been an internal error and the method
    should terminate with an error message

- update mask to contain the result of right-
  shifting msb 31 times. The value held in msb
  should not be altered.
- update foundit to contain the Boolean value FALSE
- update i to contain the value 31
- loop until i is no longer positive, executing the
  following instructions each time around the loop:
  - if the result of the bitwise AND of
    bitmap1[bitindex2] with mask is not zero
    then update foundit to contain the Boolean
    value TRUE and update bitindex1 with the
    value currently held in i and then break out
    of this loop and continue with the remaining
    instructions for this method
  - update mask to be the result of the existing
    value of mask left-shifted   once
  - update i to contain (i - 1)
- if foundit contains the Boolean value FALSE then
  there has been an internal error and the method
  should terminate with an error message

- update mask to contain the result of right-shifting msb
  31 times. The value held in msb shold not be altered.
- update foundit to contain the Boolean value FALSE
- update j to contain the value 31
- loop until j is no longer positive, executing the
  following instructions each time around the loop:
  - if the result of the bitwise AND of
    bitmap0[bitindex1 + (bitindex2 * 32)] with mask is
    not zero then update foundit to contain the Boolean
    value TRUE and update bitremainder with the value
    currently held in j and then break out of this loop

and continue with the remaining instructions for
this method

- update mask to be the result of the existing value
  of mask left-shifted once

- update j to contain (j - 1)

- if foundit contains the Boolean value FALSE then there
  has been an internal error and the method should
  terminate with an error message

- terminate this method and return as the result the
  value obtained by the calculation (((bitindex1 +
  (ibtindex2 * 32)) * 32) + bitremainder)

8. The subsidiary method "clear_bitmap"

This method takes a single argument "size" and updates
the bitmaps so that they reflect the fact that there is '
no valid memory address in pgarray[size]. There is no
return value.

- create a 32 bit integer with only the top bit set and
  all other bits being zero (i.e. 0x80000000) and store
  it in a local variable with name "msb" .

- calculate the integer value of (size / 32) and store
  the result in a local variable with the name
  "bitindex"

- calculate the remainder after size has been divided by
  32 and store the result in a local variable with the
  name "bitremainder"

- update bitmap0[bitindex] with the result of
  calculating the bitwise AND of the current value
  stored in bitmap0[bitindex] and the result of (right-
  shifting msb bitremainder times and then inverting
  every bit). The value of msb should not be altered.

- if bitmap0[bitindex] is zero then :

- update bitremainder to be the remainder after bitindex has been divided by 32

- update bitindex to be the integer result of dividing the existing value held in bitindex by 32

5 - update bitmap1[bitindex] with the result of calculating the bitwise AND of the existing value of bitmap1[bitindex] with the result of (right-shifting msb bitremainder times and then inverting all the bits). The value of msb should not be

10 altered.

- else:

  - update bitindex to be the integer result of dividing the existing value held in bitindex by 32

  - if bitmap1[bitindex] is zero then:

15 - update bitremainder to be the remainder after bitindex has been divided by 32

  - update bitindex to be the integer result of dividing the existing value held in bitindex by 32

  - update bitmap2[bitindex] with the result of
20 calculating the bitwise AND of the existing value of bitmap2[bitindex] with the result of (right-shifting msb bitremainder times and then inverting all the bits).

25 9. The subsidiary method "set_and_search_bitmap"
This method achieves two objectives: it updates the bitmaps to indicate that pgarray[size] contains a valid memory address, and it searches for the biggest set bit that represents a pgarray index less than size. It then
30 returns the appropriate index.

- pass control to method "search", passing the argument
  size; the method "search" will return an integer that
  is then stored in a local variable with the name "i"
- if size is less than or equal to MAXFIELDS, then update
  the bitmaps as follows:
  - create a 32 bit integer with only the top bit set
    and all other bits being zero (i.e. 0x80000000)
    and store it in a local variable with name "msb"
  - calculate the integer value of (size / 32) and
    store it in a local variable with name "bitindex"
  - calculate the remainder after size is divided by
    32 and store that remainder in a local variable
    with name "bitremainder"
  - update bitmap0[bitindex] with the result of
    calculating the bitwise OR of the existing
    value of bitmap0[bitindex] with the value obtained
    by right-shifting the value stored in msb
    bitremainder times (the value held in msb should
    NOT be altered)
  - update bitmap1[bitindex / 32] with the result of
    calculating the bitwise OR of the
    existing value of bitmap1[bitindex/32] with the
    value obtained by right-shifting msb by the number
    of times that is the result of calculating the
    remainder after dividing bitindex by 32 (the value
    held in msb should NOT be altered)
  - update bitmap2[(bitindex/32)/32] with the result
    of calculating the bitwise OR of the
    existing value of bitmap2[(bitindex/32)/32] with
    the value obtained by right-shifting msb by the
    number of times that is the result of calculating
    the remainder after (bitindex/32) is divided by 32

- terminate this method and return the value i

Claims

1. A data structure comprising:

    a *data storage space;*

    a first *data array* which, for each of one or *more* *cell sizes,* stores data indicating whether or not the *data storage space* includes at least one *cell* of a certain type and of the respective *size;* and

    a second *data array* which, for any said *size* for which at least one *cell* of the determined type exists, indicates a location of at least one *cell* of that *size* and type within the *data storage space.*

2. A data structure according to claim 1 in which said predetermined *cell sizes* are no more than a predetermined maximum cell size which is independent of the total *size* of the data storage space.

3. A data structure according to claim 2 in which said predetermined *cell sizes* are all *cell sizes* up to the predetermined maximum *cell size (MAX FIELDS).*

4. A data structure according to claim 3 in which the first *data array* is selected so that for a *cell size* n it is possible to determine the smallest (integer) value of p such that p>n and there exists at least one *cell* of the predetermined type of *size* p, within a time which scales with log (MAXFIELDS).

5. A data structure according to claim 3 in which the first *data array* is selected so that for a *cell size* n it is possible to determine the largest (integer) value of p

such that p<n and there exists at least one *cell* of the
predetermined type of *size* p, within a time which scales
with log (MAXFIELDS).

5   6. A data structure according to any preceding claim in
which the first *data array* is a hierarchical *data array,*
having:

        a first level which includes one or more first
· *fields,* each first *field* indicating for a respective *size*
10 of *cell* the existence or otherwise of at least one *cell*
of that *size,* said first *fields* being divided into one or
more groups, each group representing a respective range
of *cell sizes;* and

        a second level which includes one or more second
15 *fields,* each second *field* indicating for a respective
said group the existence or otherwise of at least one
cell in the respective range of *cell sizes.*

7. A data structure according to claim 6 in which the
first *data array* contains further layers above the second
20 layer, there being in each i-th layer one or more *fields,*
for each layer except the top layer, the one or more
fields are divided into one or more groups, each group
representing a range of *cell sizes,* and in the (i+ 1)-th
layer there is a *field* for each respective group of the
25 i-th layer.

8. A data structure according to claim 7 in which in each
layer, each group represents a number *offields* in the
layer below which is a multiple of eight.

9. A data structure according to any of claims 6 to 8 in
30 which each of said fields of at least the first level
indicates the existence or otherwise of at least one cell
of said size or size range by a binary variable.

10. A data structure comprising:

a *data storage space;*

a first *data array* which, for each of one or more
cell size ranges, stores data

indicating whether or not the *data storage space* includes

5   at least one *cell* of a certain type within the respective
size range; and

a second *data array* which, for any of said *size*
ranges for which at least one *cell* of

the determined type exists, indicates a location of at

10   least one of those *cells* within the *data storage space.*

11. A data structure according to any preceding claim
which includes a *field* which stores the location of at
least one of: (i) a *cell* of the predetermined type having

15   the greatest *size* for which *cells* of the predetermined
type exist, or (ii) a *cell* of the predetermined type   '
having the smallest *size* for which *cells* of the
predetermined type exist.

20   12. A data structure according to any preceding claim in
which the *cells* contain pointers to each other such that,
starting from any *cell* of any predetermined *cell size,*
any other *cell* of the predetermined type and the same
cell size can be reached by moving successively from one

25   *cell* to another according to the pointers.

13. A data structure according to claim 12 in which at
least one *cell* of each predetermined *size* contains a
pointer to one of the *cells* of the next higher occupied

30   predetermined *cell size,* and/or at least one *cell* of each
predetermined *size* contains a pointer to one of the *cells*
of the next lower occupied predetermined *size.*

14. A data structure according to claim 12 or claim 13 in which the order of the path defined by the pointers through *cells* of the same *size* encodes information about the *cells* of that *size*.

15. A data structure according to claim 12, claim 13 or claim 14 which includes at least one *cell* of the predetermined type which is of minimum *size,* has a predetermined location in the *data storage space* and contains a pointer to a *cell* of the predetermined type of the same or next higher *size.*

16. A data structure including:

a *data storage space,* the *data storage space* comprising one or more *cells* of a certain type, each said *cell* of said type containing at least one pointer to the, location in the *data storage space* of another said *cell* of said type; and

a *field* which stores as one or more pointers the location within the *data storage space* of at least one of: (i) one of said *cells* of said type which is at least as large as any other of said *cells* and of said type, or (ii) one of said *cells* of said type which is at least as small as any other of said *cells* and of said type;

said pointers being such that, starting from a *cell* of which the location is indicated in said *field,* any other of said *cells* can be reached by moving successively from one *cell* to another according to the pointers in a path in which the *size* of the *cells* varies monotonically.

17. A memory management method for controlling one or
more *data storage devices* which support a *data storage
space,* the memory management method including:

    deriving a first *data array* which, for each of one
5 or more *cell sizes,* stores data indicating whether or not
the *data storage space* includes at least one *free cell* of
the respective *size;*

    deriving a second *data array* which, for any said
size for which at least one *free cell* exists, indicates a
10 location of at least one *free cell* of that *size* within
the *data storage space;* and

    upon receiving an instruction to store a *data item,*
determining using said first *data array* an appropriate
cell size of a *free cell* for storing the *data item,* and
15 using said second *data array* to derive the location
within the *data storage space* of a location of a *free*   ᴵ
cell of the determined *cell size.*

18. A memory management method for controlling one or
20 more *data storage devices* which support a *data storage
space,* the *data storage space* containing:

    one or more *free cells,* each said *cell* containing at
least one pointer to the location in the *data storage
space* of another said *cell;* and
25   a *field* which stores the location within the data
space of at least one of: (i) one of said *cells* which is
at least as large as any other of said *cells,* or (ii) one
of said *cells* which is at least as small as any other of
said *cells;*
30   said pointers being such that, starting from a *cell*
of which the location is indicated in
said *field,* any other of said *cells* can be reached by
moving successively from one *cell* to another according to

the pointers in a path in which the *size* of the *cells*
varies monotonically;

the method including identifying a *free cell* of a
certain *size* by starting from a first free cell, and
5   moving successively from one *cell* to another according to
the pointers until a *cell* of that *size* is reached.

19. A memory management method for controlling one more
data storage devices which support a *data storage space*
10   containing one or more *zombie cells,* each *zombie cell*
containing at least one pointer to the location in the
data storage space of another *zombie cell;*

the method employing *afield* which stores the
location within the data storage space of one of said
15   *zombie cells* which is at least as large, or at least as
sl:llall, as any other of said *zombie cells;*   ,

said pointers being such that, starting from a
zombie cell of which the location is indicated in said
field, any other of said *zombie cells* can be reached by
20   moving successively from one *zombie cell* to another
according to the pointers in a path in which the *size* of
the *zombie cells* varies monotonically;

the method including freeing *zombie cells* in an
order defined by starting from a first *zombie cell* and
25   moving successively from one *zombie cell* to another
according to the pointers.

20. A memory management method for controlling one more
data storage devices which support a *data storage space*
containing one or more *cells,* each *cell* containing at
30   least one pointer to the location in the *data storage*
space of another *cell,* and each *cell* additionally being
associated with at least one integer property;

the method employing *afield* which stores the
location within the data storage space of one of said

cells which contains a *value* of said property which is at least as large, or at least as small, as any other value of said property of said *cells;*

said pointers being such that, starting from a *cell*

5   of which the location is indicated in said *field,* any other of said *cells* can be reached by moving successively from one *cell* to another according to the pointers in a path in which the value of said property of the *cells* varies monotonically;

10   the method including deleting *cells* in an order defined by starting from a first *cell,* and moving successively from one *cell* to another according to the pointers.

15 21. A *data storage device* which is arranged to store a data structure according to any of claims 1 to 16.

22. A data storage device which performs a method according to any of claims 17 to 20.

20

23. A computer system for performing a *computational task* comprising a *data storage device* according to claim 21 or 22.

25

1/7
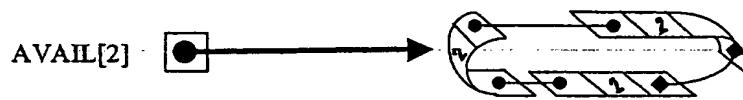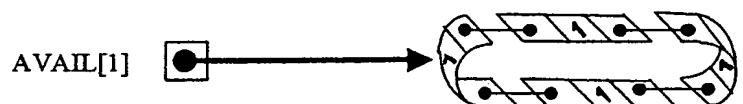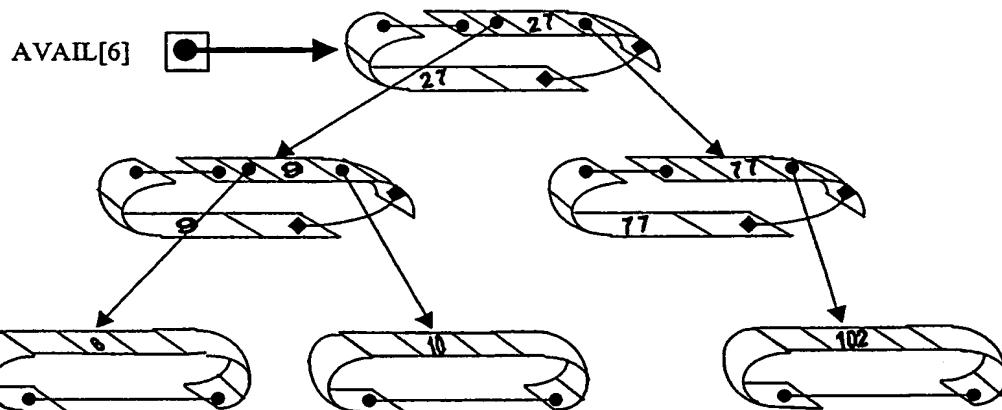
Fig. 1
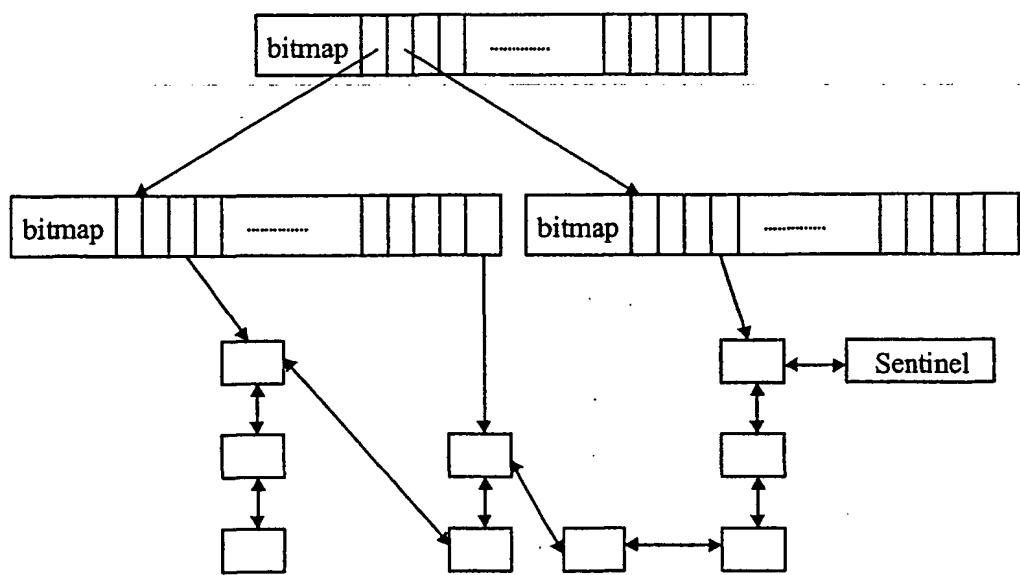
Figure 2

3/7
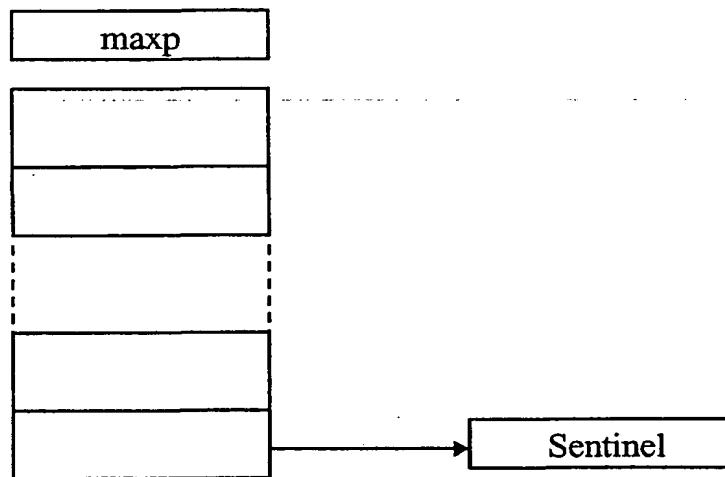
Figure 3 : Initial State of a Peer-Group List
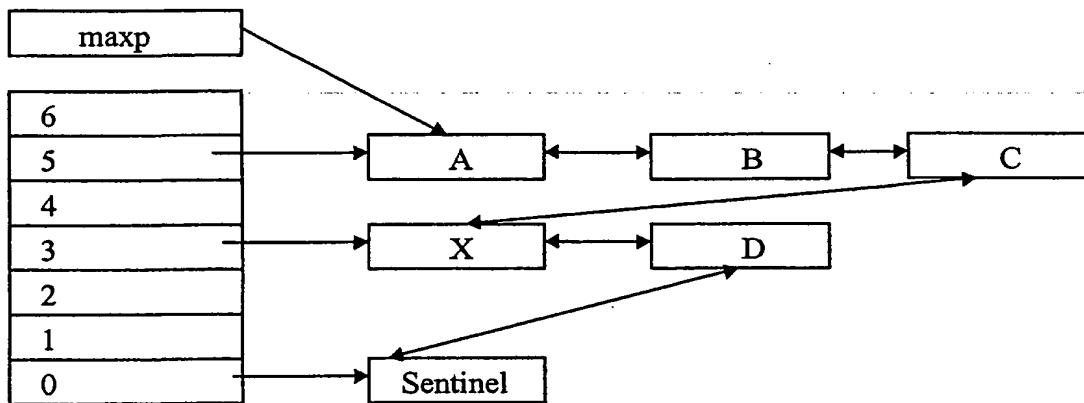
Figure 4 : Before Deletion of cell X
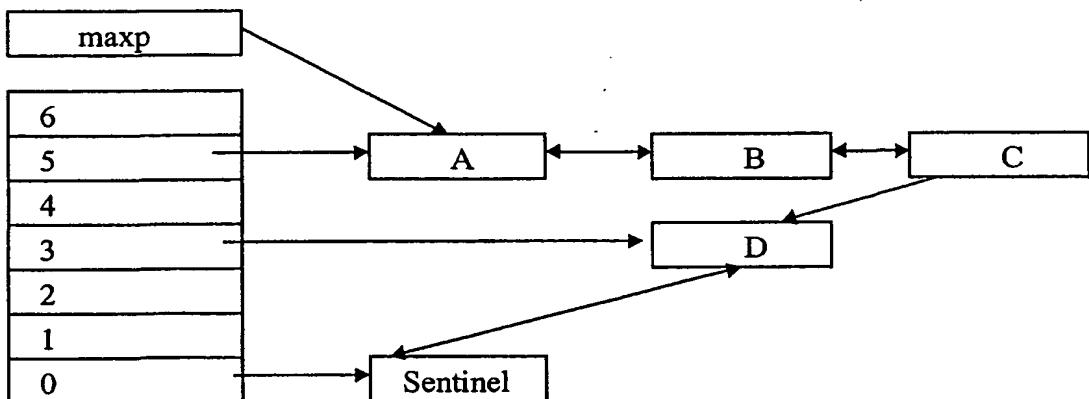
Figure 5 : After Deletion of cell X

5/7
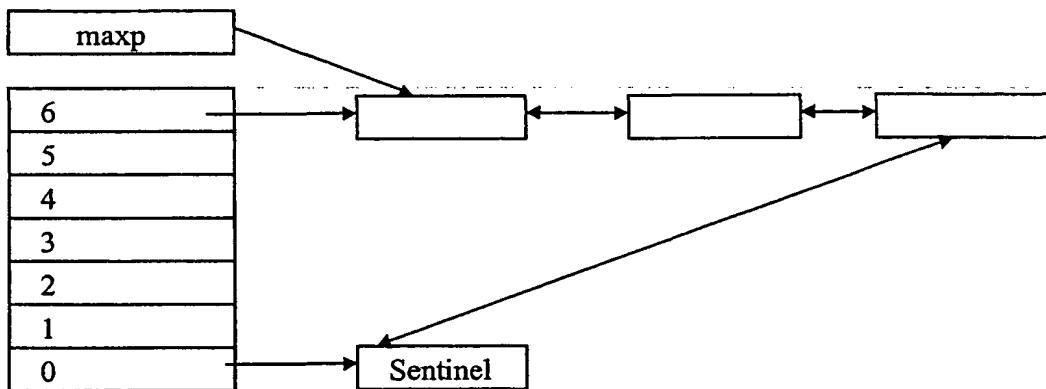
Figure 6 : Before Insertion of cell
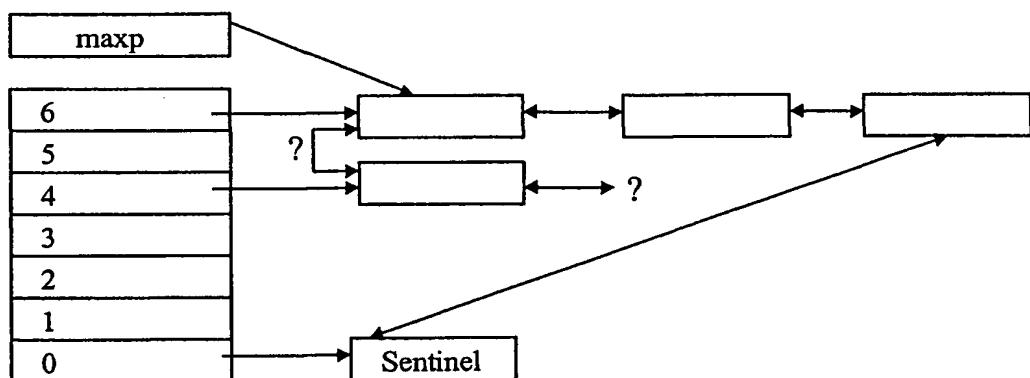
Figure 7 : A problem with Insertion

6/7
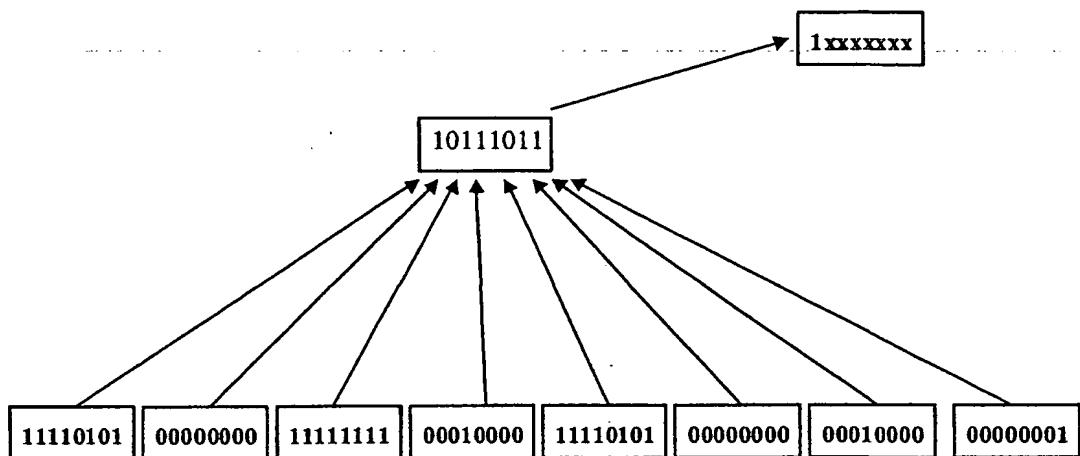
Figure 8 : Bitmap Abstractions
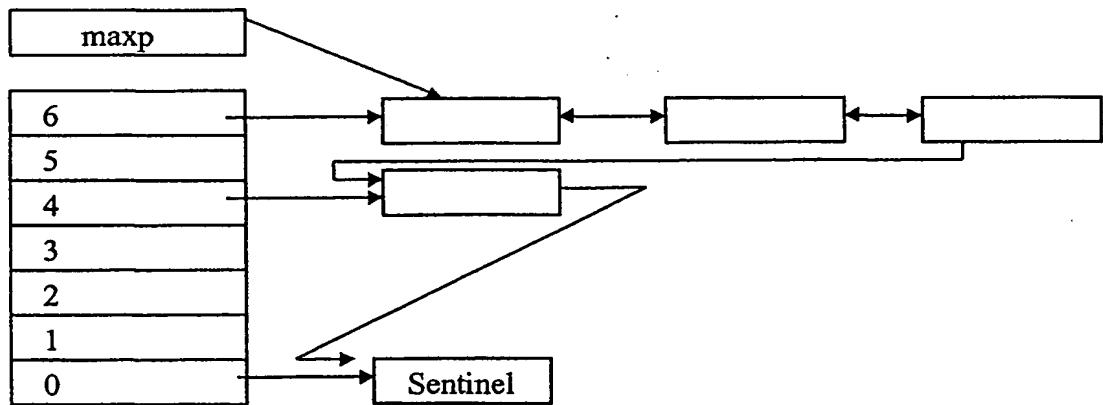

Figure 9 : Insertion problem resolved
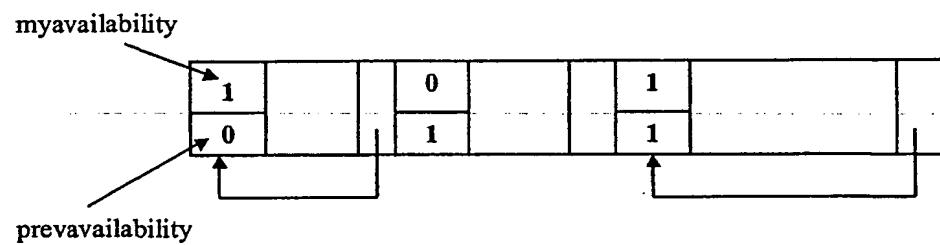
Figure 10 : Availability of Cells
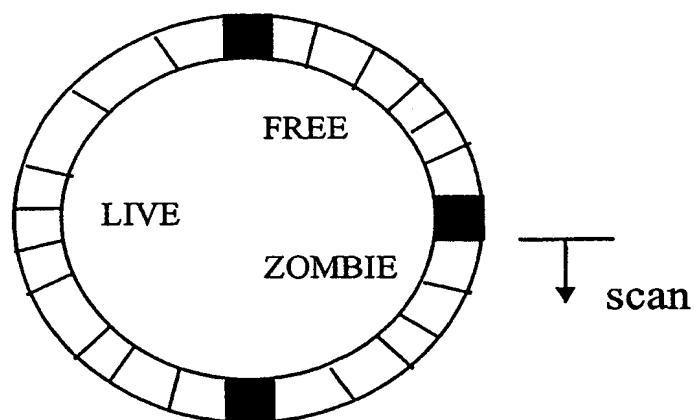
Figure 11: Treadmill Similarity